From DAGs to Riches: Improving Transaction Throughput in a Deterministic System

1st Rakrish Dhakal
CMSC624 Spring 2024
University of Maryland
College Park, MD
rakrish@umd.edu

2nd Alexander Movsesyan
CMSC624 Spring 2024
University of Maryland
College Park, MD
amovsesy@umd.edu

3rd Pranav Sivaraman
CMSC624 Spring 2024
University of Maryland
College Park, MD
psivaram@umd.edu

4th Daniel Wei
CMSC624 Spring 2024
University of Maryland
College Park, MD
dwei@terpmail.umd.edu

5th Sathwik Yannamaddi
CMSC624 Spring 2024
University of Maryland
College Park, MD
sathwik7@umd.edu

***Abstract*—Traditional transaction scheduling methods in relational databases can rely heavily on centralized schedulers and concurrency control protocols that may encounter various bottlenecks, limiting system throughput and scalability. This paper introduces a novel approach leveraging directed acyclic graphs (DAGs) of dependencies to intelligently schedule transactions while minimizing contention to avoid a scheduler bottleneck. Our proposed work, DAGER, implements DAG-based scheduling to overcome the limitations of locking, OCC, and MVCC. We explore using both a continuous scheduler, as well as an epoch-based one, inspired by the popular deterministic scheduling system Calvin. Furthermore, DAGER decentralizes the committing process and transactions from the scheduler thread, enabling further parallelism. We also implement various optimizations, including lockless atomic queues and work-stealing local queues per worker to address load imbalance. Comparative analysis shows that our DAG-based approach outperforms the original system in certain crucial conditions. This research provides a promising first step towards utilizing DAGs for efficiency improvements in transactional systems.**

I. Introduction

Achieving high throughput and scalability while ensuring determinism is challenging. In fact, adding determinism to non-deterministic systems can introduce additional complexities to avoid sacrificing throughput. Calvin [1] is a transactional database layer is an example that can be built on top of of a non-deterministic systems. This project extends the Project 2 codebase, initially developed to explore various transaction processing algorithms.

The paper details the design of our deterministic concurrency control algorithm, named DAGER, which leverages Directed Acyclic Graphs (DAGs) to track transaction dependencies more efficiently and parallelizes transaction processing across multiple worker threads. We compare two different flavors of DAGER: an epoch-based method, which batches transactions into epochs similar to Calvin, and a continuous method that foregoes the batching process.

Both modes utilize transaction dependency graphs to allow for immediate execution of non-dependent transaction. As transactions complete we can update the transaction dependencies using the DAG and immediately schedule new transactions that have no more dependencies to minimize idle time.

In addition to the scheduling logic, we explored several performance optimizations. One significant addition was the introduction of a work-stealing scheme in the thread pool. Reducing the overhead of threads in the thread pool acquiring locks and managing work can see great improvements, especially as transaction durations vary in mixed workloads.

By benchmarking the DAGER algorithm in both epoch-based and continuous implementations against other algorithms from the Project 2 codebase, our work aims to achieve significant improvements in transaction throughput. The results not only confirm the advantages of our approach but also highlight the practical benefits

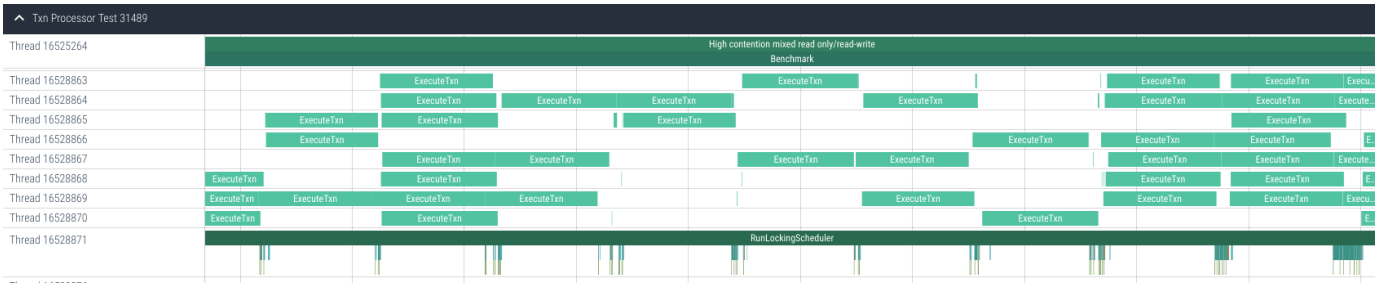


Fig. 1. Perfetto trace of Locking B from Assignment 2. We see that worker threads are sitting idle for substantial periods of time.

in deterministic processing environments and provide insights into the trade-offs of batch versus continuous processing strategies.

Section II describes the project upon which we built our work. In Section III, we survey previous works that have used dependency graphs for scheduling work. Section IV delves into our implementation in more depth, and we present our results in Section V. Finally, we conclude with a brief outlook on possible future work in Section VII.

II. Background

Our work builds on top of the code-base for CMSC624 Assignment 2 [2], which has students implement the following concurrency control schemes in a simplified, in-memory key-value store:

- **Locking A**: strict-locking with reads and writes holding exclusive locks, using a single-threaded lock manager
- **Locking B**: strict-locking with shared locks for reads and exclusive locks for writes, using a single-threaded lock manager
- **Optimistic concurrency control (OCC)**: non-locking, lets transactions proceed optimistically with a validation phase to ensure serializability. Includes serial and parallel validation.
- **Multiversion concurrency control (MVCC)**: non-locking, creates copies of records at different points in time to avoid contention, exploiting auxiliary memory space

A transaction is implemented as a struct, whose read-set and writeset are known before it is scheduled. Since our work builds upon this project, we do not support transactions that are fractional or whose readsets and writesets are not known ahead of time.

Since the aim of this work is to increase the throughput achieved by these methods, we compare our methods and results we periodically refer to these schemes throughout the paper as Locking-A, Locking-B, OCC, OCC-P, and MVCC.

To identify potential bottlenecks in Assignment 2, we collected a trace of the Assignment 2 benchmark run using Perfetto [3]. As seen in Figure 1, we observe that there are many times where the threads in the thread pool were idle. In order to maximize throughput we would need to minimize the idle time of threads in the thread pool. We identified two major bottlenecks in the current implementation.

A. Scheduler bottleneck

One of the main bottlenecks we identified in Assignment 2 is that the current scheduler, which is single-threaded, is tasked with the work of processing each incoming transaction, assigning transactions to a thread in the thread pool, and then eventually validating every completed txn and adding more ready transactions. There are spans of time in Figure 1 where the scheduler is busy with either validating txns or processing txns, which leads to idle worker threads in the thread-pool.

B. Round-robin work assignment

We also discovered that Assignment 2 implements its thread pool in a separate class called *StaticThreadPool*. StaticThreadPool implements a function, *AddTask*, which receives a function for a worker thread to complete. More specifically, the function is *ExecuteTxn*, which contains a reference to a specific transaction for a worker thread to execute. StaticThreadPool also contains a separate local queue per worker, which are populated in a round-robin fashion. While round-robin is simple to implement, it is not the best load balancing scheme, and could lead to certain worker threads performing more work than others, reducing concurrency and throughput.

III. Related Work

At the heart of this project is a deterministic concurrency control protocol. These types of protocols ensure consistent and predictable execution of transactions,

guaranteeing that given the same inputs and execution sequence, transactions will produce the same result every time. The system architecture of this project is mostly inspired by Thomson et al.'s Calvin [1], which features the same basic sequencer and scheduler layers in order to facilitate determinism. The sequencer is responsible for determining the global serial order, while the scheduler employs a deterministic locking scheme to ensure that serial order is obeyed in execution. While a rudimentary version of the sequencer and scheduler were already implemented for us, we sought to improve upon the performance of this implementation by eliminating a few glaring bottlenecks and making better use of concurrency.

A core component of the work presented in this paper is the separation of concurrency control and execution. Faleiro & Abadi present BOHM [4], a concurrency protocol primarily geared toward main-memory multi-versioned database systems. It achieves excellent performance and scalability while maintaining serializable execution by separating concurrency control logic from transaction execution logic. More specifically, the concurrency control logic works to determine the proper serialization order, while also constructing a data structure to allow the execution logic to run independently of concurrently running transactions. This greatly increases concurrency and also helps to alleviate some scalability bottlenecks. However, it is important to note that such a plan requires prior knowledge of each transaction's write-set (though pre-declaration is not necessary, as a a speculative technique predicting write-sets is feasible). This fact is also true for our project.

There have been many attempts to increase the amount of concurrency being leveraged inside deterministic concurrency control algorithms, and the use of DAGs to construct dependency graphs in order to facilitate more concurrency is not a novel idea. Whitney et al. proposes an algorithm called OBEYORDER [5] which enforces a serializable execution that is equivalent to the arrival order of transactions to the system. After logging a batch of transactions on disk, OBEYORDER constructs a graph in which directed edges are drawn from earlier transactions to later transactions in the batch if they conflict. The roots of the graph, representing transactions with zero concurrent conflicts, are executed in parallel and then removed. This process is repeated until no nodes remain in the graph. This allows the transactions in the batch to be executed with respect to the order implied by the edges while also maintaining consistency. This usage of dependency graphs serves as the basis for increasing parallelism within this project.

Tariq et al. present a very similar use of DAGs in a different context [6], employing them for task scheduling in heterogeneous computing environments. Instead of representing transaction conflicts, DAGs are constructed to model task precedence constraints and execution order, ensuring that tasks are processed only after their predecessors are complete. The scheduling algorithm is then able to efficiently map tasks to various processors, prioritizing them based on computation and communication costs to minimize the overall makespan—the total time required to complete all tasks. This method not only adheres to task precedence and optimizes resource utilization but also strategically handles dynamic scheduling challenges in distributed systems.

Huang et al.'s Taskflow leverages the same general task graph-based approach to simplify writing parallel and heterogeneous task programs in modern C++ [7]. In a similar manner to Huang et al.'s work, Taskflow uses DAGs to efficiently manage and schedule tasks by representing them as nodes and their dependencies as directed edges. This structure enables Taskflow to identify independent tasks that can be executed in parallel, maximizing computational resource utilization and reducing execution time. Tasfklow then uses dynamic scheduling based on the DAG to ensure that tasks are executed in the correct order, taking into account computation and communication costs to optimize the makespan. This approach allows Taskflow to scale effectively across heterogeneous computing environments, handling various workloads and adapting to different hardware capabilities.

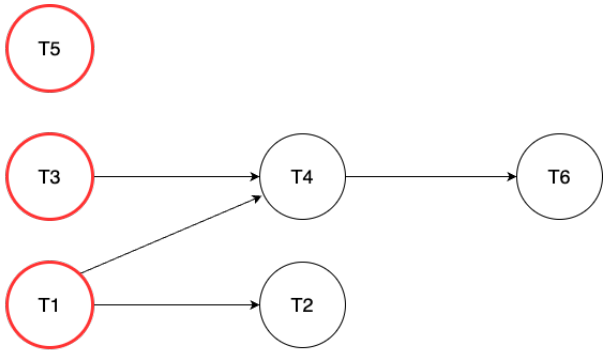
The concept of dependency graphs is also used by Faleiro et al. to propose a lazy transaction execution engine [8], in which transactions can be considered durable after being partially executed. In order to facilitate lazy execution, the concept of an uncomputed, yet extant, record called a "sticky" is used. In order to substantiate this sticky, the system uses a DAG to track and execute in order all transactions on whose write-sets the sticky's corresponding transaction transitively depends. While the application of the DAG is quite different from that of this paper, its core concepts remain the same.

IV. Implementation

In this section, we describe the implementation details of DAGER'S epoch-based sequencer and scheduler, the continuous scheduler, and further optimizations that we have made on Assignment 2 to improve the bottlenecks described in section II.

To address the scheduler bottleneck identified in Section IV, our work moves a lot of the work that the scheduler performs to the worker threads. More specifically, DAGER lets each worker thread validate its transactions, perform writes and commits, and schedule further transactions. This frees work from the single-threaded scheduler, which can easily become a bottleneck since it has to perform these tasks serially.

To maintain serializability among transactions, DAGER intelligently schedules each transaction only (and immediately) when that transaction is allowed to be executed. More specifically, the scheduler component constructs a dependency graph of incoming transactions, where each transaction is represented as a node. Then, the scheduler and worker threads work together to schedule transactions that do not have any dependencies. A *dependency*, represented by a directed edge from transaction-node T_1 to transaction-node T_2, means that T_2 should be executed strictly after T_1 to maintain serial order. Since there is an edge from T_1 to T_2, we consider T_2 a *neighbor* of T_1. When a transaction does not have any dependencies, the number of incoming edges to its node (referred to as the *in-degree*) is zero, and the transaction is ready to be executed.



Transaction	Read Set	Write Set
T1	A	B
T2	B	C
T3	A	D
T4	B, D	E
T5		F
T6	E	G

Fig. 2. Example dependency graph of transaction-nodes created by our DAG-construction algorithm.

A. Constructing the dependency graph

To construct the directed acyclic graph (DAG) of transaction dependencies, we maintain for each key:

1) **shared_holders**: A set of transactions that have a shared ”lock” for that key.
2) **last_excl**: The latest transaction with an exclusive ”lock” for that key.

When a transaction arrives, the scheduler traverses its readset and writeset, and inserts the proper edges from transactions in *shared_holders* and *last_excl* to the current transaction. Then, we update the values above for the affected keys to the current transaction. Since the scheduler ensures that the system is deterministic, we only need to know which transactions come immediately before instead of storing the entire list of dependencies that hold a lock on a key, as with Locking A and Locking B from assignment 2.

Once a transaction’s readset and writeset are traversed, the scheduler immediately enqueues transactions with an in-degree of zero for the workers. Once a worker executes a transaction, it atomically updates the in-degrees of all of the neighbors of that transaction. Once a neighbor’s in-degree is zero, this means it no longer has dependencies, and can be enqueued for execution.

In Figure 2, we can see an example of the dependency graph that DAGER creates. The transaction-nodes highlighted in red have an in-degree of zero, and are thus enqueued for the workers to execute immediately.

B. Epoch Separated Execution

The Calvin system employs the notion of epochs to batch transactions by the time of their initial request. While the implementations of epochs proves useful in the distributed database space, the advantages of epochs are debatable when dealing with single node shared everything databases.

As mentioned previously, incoming transaction requests are added into a dependency DAG. If incoming transactions are separated and grouped into epochs, the DAG creation and insertions can be completed per epoch, leading to efficient lockless algorithms to schedule transaction executions. However, the separation of transactions into epochs guarantees that transactions in disjoint epochs will not be executed concurrently regardless of their memory accesses. Thus, the usage of epochs in the DAG based scheduling is debatable in terms of theoretical performance results.

As illustrated in Figure 3, our epoch-based execution system can be separated into 3 different concurrently executing components.

1) The Sequencer: When receiving new transaction requests, we need to assign it to an epoch. The sequencer

takes in new transaction, adding it to the current epoch, then labels the epoch as ready once the predetermined epoch time has been met.

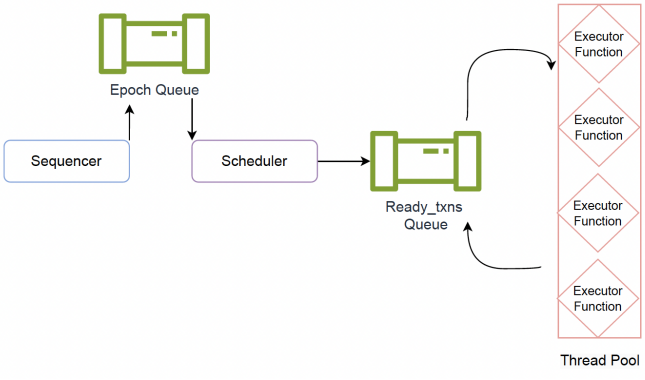


Fig. 3. System architecture of DAGER (epoch execution)

Algorithm 1 The Epoch Sequencer

```
global variables
    current_epoch, Global std :: queue
    epoch_queue, Global AtomicQueue < Epoch >
    previous_time, Global time
end global variables
procedure SEQUENCER(Txn * txn)
    current_epoch.push(txn)
    current_time ← Time()
    if (current_time − previous_time) ≥ 5 then
        previous_time ← current_time
        epoch_queue.push(current_epoch)
        current_epoch ← {}
    end if
end procedure
```

2) The Scheduler: While the Sequencer is pushing ready epochs to the global queue, the Scheduler pops from that queue and creates the dependency graph of the new epoch. Once this DAG is created, the zero-indegree transactions are added to the thread pool to execute.

3) The Executor Function: The executor function, run by each of the worker threads, continuously scans the queue of ready transactions. Once it finds a transaction, it can pop it from the queue and execute it. Since the queue only contains transactions without dependencies, it is guaranteed that executing the popped transaction at this point maintains the serial order determined by the sequencer.

Then, the worker thread independently commits or aborts the transaction without returning it back to the scheduler, which enables this to occur in parallel unlike in Assignment 2. Finally, the worker updates the in-degrees of neighbors. If a neighbor has an in-degree of zero as a result of executing this transaction, the worker adds the neighbor to the ready transactions queue.

Algorithm 2 The Epoch Scheduler

```
start sequencer thread
while not stopped do
  if epoch_queue.Pop(&curr_epoch) then
    initialize shared_holders and last_excl as empty maps
    dag ← allocate new EpochDag
    initialize adj_list, indegree, and root_txns as empty maps/queues
    while curr_epoch not empty do
      txn ← curr_epoch.pop()
      initialize adj_list[txn] as empty set
      indegree[txn] ← 0
      for all key in txn.readset do
        add txn to shared_holders[key]
        if key in last_excl and txn not in adj_list[last_excl[key]] then
          add txn to adj_list[last_excl[key]] and inc indegree[txn]
        end if
      end for
      for all key in txn.writeset do
        if key in shared_holders then
          for all conflicting_txn in shared_holders[key] do
            if txn not in adj_list[conflicting_txn] then
              add txn to adj_list[conflicting_txn] and inc indegree[txn]
            end if
          end for
          clear shared_holders[key]
        end if
        last_excl[key] ← txn
      end for
      if indegree[txn] is 0 then
        push txn to root_txns
      end if
    end while
    set dag properties to adj_list, indegree, and root_txns
    push dag to epoch_dag_queue
  end if
end while
```

C. Continuous Execution

While the epoch-separated execution model is straightforward and provides several benefits, grouping transactions into epochs has the critical drawback that epochs must be executed in serial order. In other words, transactions in a subsequent epoch that have no conflicts with transactions from the previous epoch must necessarily wait for the previous epoch to complete, which sacrifices the potential amount of concurrency (and thus throughput) that a system can achieve. As a result, we have also implemented a continuous scheduler in DAGER.

Unlike the epoch-based execution, the continuous execution does not have a sequencer – the only component

of it is the continuous scheduler (as seen in Figure 4. This scheduler, rather than building a new DAG for an epoch, modifies the global DAG in an online manner to include each transaction as it arrives.

The continuous scheduler, like the epoch scheduler, determines the dependencies of each transaction as it arrives. However, a major difference is that edges should only be added between *incomplete* transactions and the current transaction. In the epoch-based scheduler, it is guaranteed that no dependency has been executed yet – however, for a continuously-updating DAG, this is not the case. Therefore, we take extra care to only draw edges between *scheduled, but not committed* transactions and the current transaction.

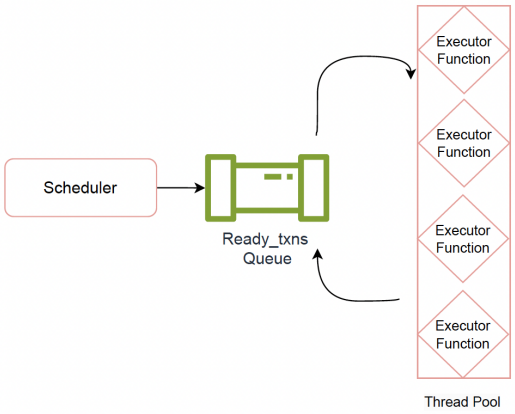


Fig. 4. System architecture of DAGER (continuous execution)

Like the epoch-based scheduler and executor, the continuous scheduler and executor also continuously enqueue transactions with indegrees of zero. The scheduler only *increments* indegrees, as it determines the dependencies of a transaction, whereas the executor only *decrements* indegrees, as it has completed a dependency.

This continuous execution model was particularly challenging to implement, primary due to the need to acquire locks on the adjacency list of transactions. The scheduler may try to add to a transaction's neighbors list while the transaction is executing, which could possibly cause neither the scheduler, nor the workers, to schedule a transaction. To make this task easier, we have further divided the continuous scheduler into two: one that locks entire DAG, and one that locks individual transaction-nodes within the DAG.

1) Global Locking: Initially, DAGER had implemented the DAG using two data structures that are typically used for performing a breadth-first-search topological sort:

1) **adj_list**: A hashmap that stores the set of neighbors (its dependents) of each transaction.
2) **indegrees**: A hashmap that stores the in-degree of each transaction (number of dependencies at any time).

Naturally, we started out re-using these data structures for the continuous scheduler. However, we determined that this requires us to lock the entire adjacency list and indegree hashmaps each time the scheduler appends to the DAG, and each time a worker thread processes a transaction. A lock needs to be held to ensure that these components do not modify the neighbors of a transaction, and that indegrees are updated atomically, guaranteeing that each transaction actually gets pushed to the ready queue.

2) Individual Transaction Locking: Locking the entire global adjacency list and indegree counts reduces throughput between sets of transaction dependencies that do not depend on each other. To exploit finer-grain locking, in this version, we instead store the adjacency list and indegree of each transaction within the transaction struct itself (as described in Section II), as opposed to globally. This way, both the scheduler and worker can lock only the necessary adjacency list and indegree values within a transaction-node, rather than locking the entire DAG. In theory, this should dramatically increase concurrency.

D. Further optimizations

1) Work Stealing: As described in Section II, Assignment 2 employs round-robin assignment of the *ExecuteTxn* function to the local queue of each worker. We have implemented a more aggressive work-stealing approach. Instead of each worker thread in the pool spin waiting for more work from its own worker queue, it instead loops through the other workers' queues and attempts to steal from the front of another worker's queue. This can be especially beneficial in mixed transaction workloads where one worker may be assigned long-running transactions and a bunch of smaller transactions. Other idle worker threads can then steal the short-lived transactions, allowing for increased throughput.

V. Evaluation

To investigate the performance of DAGER, we performed the benchmark that is provided with Assignment 2 with a few modifications. The benchmark consists of the a series of simulations performed under the following workloads:

Algorithm 3 Executor function for continuous execution (with individual transaction locking)

```
txn ← NULL
while not stopped do
    if calvin_ready_txns.Pop(txn) then
        ExecuteTxn(txn)
        if txn.Status() = COMPLETED_C then
            ApplyWrites(txn)
            committed_txns.Push(txn)
            txn.status ← COMMITTED
        else if txn.Status() = COMPLETED_A then
            txn.status ← ABORTED
        else
            DIE("Completed Txn has invalid TxnStatus: " + txn.Status())
        end if
        txn.neighbor_mutex.lock()
        neighbors ← txn.neighhbors
        txn.neighbor_mutex.unlock()
        for nei in sorted_neighbors do
            nei.indegree_mutex.lock()
            nei.indegree ← nei.indegree − 1
            if nei.indegree = 0 then
                calvin_ready_txns.Push(nei)
            end if
            nei.indegree_mutex.unlock()
        end for
        txn_results.Push(txn)
    end if
end while
```

1) Low-contention read-only (5 and 30 records)
2) High-contention read-only (5 and 30 records)
3) Low-contention read-write (5 and 30 records)
4) Low contention read-write (5 and 10 records)
5) High contention read-write (5 and 10 records)
6) High contention mixed read only/read-write

Each of the above workloads are simulated for two rounds, and their results are averaged. In each of these cases, transactions of varying durations (0.1ms, 1ms, and 10ms) are simulated using a busy loop. While this helps us fix transaction durations and observe theoretical performance under the different schemes, it does not necessarily simulate real-world transactions with complexity, and assumes a uniform distribution of contention within the different workloads. However, since our primary goal was to increase throughput, we have chosen to use the Assignment 2 benchmark to provide a comparative analysis against the existing schemes.

We have modified the Assignment 2 benchmark, which consists of *Serial*, *Locking A*, *Locking B*, *OCC*, *OCC-P*, and *MVCC* (detailed in Section II), to include our epoch-based execution (*DAGER-E*), continuous execution with global locks (*DAGER-C*), and continuous execution with individual transaction locks (*DAGER-I*). The experiment was run on a single compute node of a high-performance cluster using an AMD EPYC 7763 processor. We now present the results of this experiment, and discuss its implications for DAGER.

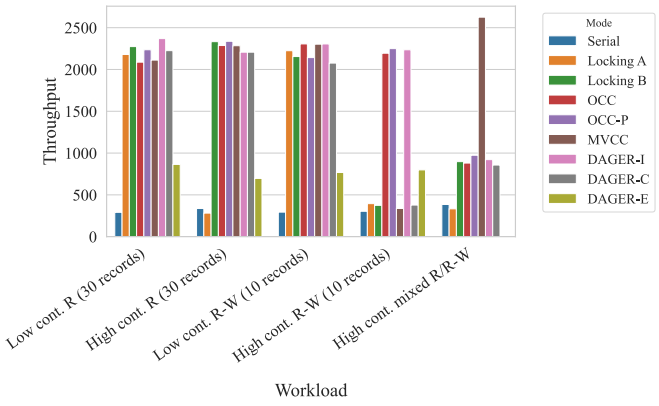


Fig. 5. Our benchmark results compared against schemes from Assignment 2.

A. DAGER compared to other Concurrency Control Schemes

As seen in Figure 5, DAGER-I demonstrates generally good performance compared to other concurrency control models, particularly in mixed workloads. When directly compared with Locking B, another deterministic database control scheme, DAGER-I consistently matches or even surpasses it in throughput. However, MVCC remains the top-performing model for read and write scenarios where reads predominate. This is because MVCC can efficiently store multiple data versions and serve short reads irrespective of longer, larger transactions.

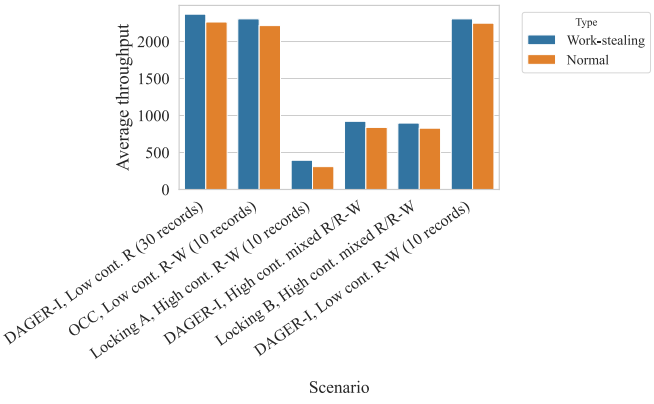


Fig. 6. Best-performing cases for our work-stealing optimization.

B. Comparison of different DAGER algorithms

DAGER-I consistently outperforms DAGER-C, with DAGER-E consistently being the worst among them. The major reason for the performance gap in DAGER-I would be that we have finer-grained locking compared to DAGER-C. In DAGER-C, we have global locking to manage the access to and from the global adjacent list and indegree array. In DAGER-I, we are able to lock individual mutexes for the adjacency list and the indegree node counts, allowing for less contention among the locks and enabling more throughput. The poor performance of DAGER-E is likely due to the need for a subsequent to wait for a previous epoch to fully finish sacrifices concurrency and therefore throughput. In addition, the delay of DAGER-E in preparing each sequencer batch to send to the scheduler, further delaying the creation of the DAG, which is then sent to the worker threads.

C. Work Stealing vs Round Robin

The effect on our aggressive work stealing algorithm is quite minimal as show in Figure 6. However, for long database workloads and mixed transaction lengths this small difference can accumulate over time, resulting in overall higher throughput.

VI. Future Work

Our work on DAGER presents a promising first step towards leveraging directed acyclic graphs (DAGs) for improved transaction scheduling. There are several modifications of concurrency control we believe could improve our algorithm's performance

One direction is to implement finer-grained locking. By reducing the granularity of locks to individual data entries, we can decrease contention. Techniques such as transactional memory or lock-free data structures can be explored as well. Lock-free queues can help eliminate bottlenecks caused by lock contention, allowing multiple threads to enqueue/dequeue concurrently without blocking. Additionally, developing a approach to manipulating the DAG with lock-free structures that allow concurrent updates could improve the performance of DAGER. The current epoch-based scheduler can be a bottleneck in comparison to the sequencer since it is single-threaded, so another direction for improvement could be a multi-threaded scheduler which could reduce the construction time of the DAG strucuture.

VII. Conclusion

According to our benchmarks, while our continuous-scheduling mode occasionally outperforms in specific scenarios, its overall performance was lacking compared to the original algorithms. The continuous-scheduling mode has potential and could outperform traditional methods with further optimizations. On the other hand, the epoch mode of DAGER repeatedly lagged behind the performance of other methods. This performance gap highlights the current trade-offs of batching transactions into epochs, which limits concurrency in sequential execution of epochs. Despite this, the epoch-based approach provides a framework for further exploring DAG-based scheduling in a deterministic environment.

Overall, our findings suggest that the use of DAGs is a viable research path and it is worth further research to fully realize the potential of DAGs in improving concurrency control and throughput in transactional systems.

References

[1] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1–12. [Online]. Available: https://doi.org/10.1145/2213836.2213838

[2] umd-db, *Assignment 2: Concurrency Control Schemes*, 2024. [Online]. Available: https://github.com/umd-db/cmsc-624-spring24

[3] "Perfetto - system profiling, app tracing and trace analysis," https://perfetto.dev/docs/, 2022, accessed: 2023-04-01.

[4] J. M. Faleiro and D. J. Abadi, "Rethinking serializable multiversion concurrency control," *Proc. VLDB Endow.*, vol. 8, no. 11, p. 1190–1201, jul 2015. [Online]. Available: https://doi.org/10.14778/2809974.2809981

[5] A. Whitney, D. Shasha, and S. Apter, "High volume transaction processing without concurrency control, two phase commit, sql or c++," 1997.

[6] R. Tariq, F. Aadil, M. F. Malik, S. Ejaz, M. U. Khan, and M. F. Khan, "Directed acyclic graph based task scheduling algorithm for heterogeneous systems," in *Intelligent Systems and Applications: Proceedings of the 2018 Intelligent Systems Conference (IntelliSys) Volume 2*. Springer, 2019, pp. 936–947.

[7] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1303–1320, 2021.

[8] J. M. Faleiro and D. J. Abadi, "Rethinking serializable multiversion concurrency control," *arXiv preprint arXiv:1412.2324*, 2014.