Everything is a Transaction: Unifying Logical Concurrency Control and Physical Data Structure Maintenance in Database Management Systems

Ling Zhang, Matthew Butrovich, Tianyu Li♠, Yash Nannapanei♦
Andrew Pavlo, John Rollinson♣, Huanchen Zhang⋆
Ambarish Balakumar, Daniel Biales, Ziqi Dong, Emmanuel Eppinger, Jordi Gonzalez
Wan Shen Lim, Jianqiao Liu, Lin Ma, Prashanth Menon, Soumil Mukherjee, Tanuj Nayak
Amadou Ngom, Jeff Niu, Deepayan Patra, Poojita Raj, Stephanie Wang, Wuwen Wang
Yao Yu, William Zhang
Carnegie Mellon University
♠Massachusetts Institute of Technology, ♦Rockset, ♣Army Cyber Institute ⋆Tsinghua University,
lingz2@cs.cmu.edu

ABSTRACT

Almost every database management system (DBMS) supporting transactions created in the last decade implements multi-version concurrency control (MVCC). But these systems rely on physical data structures (e.g., B+trees, hash tables) that do not natively support multi-versioning. As a result, there is a disconnect between the logical semantics of transactions and the DBMS's underlying implementation. System developers must invest in engineering efforts to coordinate transactional access to these data structures and non-transactional maintenance tasks. This burden leads to challenges when reasoning about the system's correctness and performance and inhibits its modularity. In this paper, we propose the Deferred Action Framework (DAF), a new system architecture for scheduling maintenance tasks in an MVCC DBMS integrated with the system's transactional semantics. DAF allows the system to register arbitrary actions and then defer their processing until they are deemed safe by transactional processing. We show that DAF can support garbage collection and index cleaning without compromising performance while facilitating higher-level implementation goals, such as non-blocking schema changes and self-driving optimizations.

1. INTRODUCTION

Race conditions and transaction interleavings within an MVCC DBMS remain challenging implementation hurdles despite insights from decades of development [9, 24, 22, 28, 20]. This difficulty partly arises because of a disconnect between the concurrency control semantics of the *logical* layer of the system (e.g., transactions, tuples) and the synchronization techniques of the underlying *physical* data structures (e.g., arrays, hash tables, trees). Developers must carefully reason about races within these physical objects and devise bespoke solutions that are transactionally correct and scalable.

The core challenge in this is to coordinate two types of accesses to the same physical data structures: (1) transactional runtime operations (e.g., inserting a key into an index) and (2) non-transactional maintenance tasks (e.g., removing invisible versions from an index). The DBMS can simplify this dichotomy by unifying them under the same transactional semantics. Under this model, the system uses transactional timestamps as an epoch protection mechanism to prevent races between maintenance tasks and active transactions [26]. For example, an entry in an MVCC version chain is obsolete when it is no longer visible by any active transactions in the system. The DBMS's garbage collector, therefore, looks up the oldest running transaction in the system and safely removes the entries created before that transaction starts [15, 7].

In this paper, we generalize this idea into a framework, called the **Deferred Action Framework** (DAF), that can process maintenance tasks in a safe and scalable way. We integrate DAF into a DBMS's transaction processing engine and provide a simple API for *deferring* arbitrary *actions* on physical data structures. Specifically, DAF guarantees to process actions deferred at some timestamp t only after all transactions started before t have exited. This provides epoch protection to transactions and maintenance tasks, without requiring a separate mechanism for refreshing and advancing epochs. Unlike other epoch-based protection implementations [26], DAF satisfies complex ordering requirements for actions deferred at the same time through a novel algorithm of repeated deferrals. This enables DAF to process maintenance tasks in parallel while satisfying any implicit dependencies between them (e.g., delete a table only after all version chain maintenance on the table is finished).

To evaluate DAF, we integrated it into the **NoisePage** [1] DBMS to process two internal tasks: (1) MVCC version chain maintenance and (2) index maintenance. We found that DAF reduces the implementation complexity of these tasks while offering competitive performance compared to hand-optimized alternatives. DAF also helps us reason about more complex transaction interleavings in databases with evolving schemas, and it serves as a basis for supporting non-blocking schema changes. Because DAF decouples action processing from action generation, it gives us flexibility to dynamically adjust the action processing strategies. Additionally, DAF functions as a central point for runtime metrics collection in our system. These features combine to make DAF a strong building block for the self-driving [21] features in NoisePage.

The rest of this paper is organized as follows. We begin in Section 2 with a survey of existing solutions for physical data structure synchronization and epoch-based protections. Section 3 then presents DAF's programming model. We show the correctness of DAF and address the implementation challenges in Section 4. Section 5 details other uses of DAF within NoisePage. We present our experimental evaluation of DAF in Section 6 and conclude with a summary of related and future works in Sections 7 and 8.

2. BACKGROUND

The crux of MVCC is that a writer to a tuple creates a new "version" instead of taking a lock and performing in-place updates [6]. Under this scheme, readers can access proper older versions without being blocked by writers. Such scalability benefits come at the cost of additional storage overhead and implementation complexity that systems need to address. Systems need to store and differentiate multiple versions, maintain them until no transaction can access them, and quickly discard them to free up storage space after that. In this section, we provide an overview of these challenges and a brief survey on existing solutions. We also present the MVCC implementation of NoisePage that DAF integrates with.

2.1 Data Structure Maintenance in MVCC

The need to keep track of multi-versioning information permeates across many of an MVCC DBMS's internal data structures. For example, the DBMS organizes tuples in *version chains* that are linked lists of all physical versions of a single logical tuple [28]. But indexes may have multiple references pointing to the same underlying tuple if the versions of that tuple differ in the attribute indexed. To support non-blocking schema changes [19], the system additionally needs to accommodate multiple versions of the schema co-existing in the system. With multi-versioned schemas, however, the DBMS needs to support multiple entries in its plan cache to support transactions with different visibilities.

There are different ways to solve this multi-versioning data structure problem. The system can assign the responsibility of maintenance to either a set of background threads and require transaction processing threads to cooperatively perform the duty or a combination of the two [9]. Additionally, data structure maintenance happens concurrently with user transactions, and the maintenance tasks must coordinate access to internal data structures with the transactions for memory safety and semantic correctness. Such coordination must be scalable, so as to not affect the transactional performance of the system. Many systems resort to an epoch-based protection mechanism to achieve this [26, 8, 12]. Under this scheme, transactions protect against concurrent maintenance by registering with a monotonically increasing counter, or epoch, and deregister once they no longer require protection. The epoch steadily advances, so eventually, an epoch has no registered transactions and becomes unprotected. The maintenance tasks associated with that epoch can then be safely processed without interfering with running transactions. To our knowledge, all of these systems implement epoch protection as a stand-alone component that does not integrate into the transactional semantics of MVCC. Thus, DBMS developers maintain the epoch explicitly in the transactional processing code.

3. FRAMEWORK OVERVIEW

In this section, we present an overview of the logical model of DAF and its programming interface. At the core of DAF is the concept of *actions* that are internal operations that DBMS performs on physical data structures, such as pruning a version chain, compacting a storage block, or removing a key from an index. The system executes actions in response to user requests, but the execution must be deferred until safety requirements are met. For example, when a query updates a tuple, the system can remove the older version only after that version is no longer visible to any current or future transactions in the system.

DAF exposes a single API to the rest of the system: $defer\ (action)$. This function takes in an action as its input parameter and tags it with the current timestamp. DAF guarantees to invoke the given action if and only if there are no transactions in the system with a start timestamp *smaller* than the tagged timestamp.

The action is a lambda function that contains the references to the physical data objects that it will modify or deallocate.

3.1 System Characteristics

To simplify the implementation of DAF, we require the DBMS to have certain properties. In particular, the DBMS must use timestamp-ordering for MVCC and track the begin timestamp of all transactions that may still be holding references to objects in the system. The system must provide a method for transactions to query the begin timestamp of the oldest active transaction. In addition, a transaction must also be able to obtain an "observable" timestamp: the timestamp at which all active transactions are guaranteed to see its logical effects. While the discussion below will further assume that the system uses a single, global counter for timestamps, the framework functions correctly as long as the timestamp returned by "next" is guaranteed to be logically after any timestamps already assigned to transactions. We implemented DAF in NoisePage, but several other modern in-memory databases meet these requirements or could support it with minor modifications to expose the necessary functionality in their timestamp systems.

For the rest of the paper, we make only minimal assumptions about the threading model of the system. In particular, we do not make any assumption about whether the threading model is implemented in hardware, the kernel, or userspace. For simplicity, however, we assume that the unit of work for a thread is a *task*: it will execute a single task to completion before starting another. We will further distinguish between two types of tasks: *worker tasks* which execute individual transactions in the system and *action tasks* which are maintenance routines which cleanup and release resources no longer in use.

3.2 Implementation

Figure 1 presents an overview of how DAF interacts with the system's transaction manager. In the simplest configuration, DAF utilizes a single action queue and a dedicated thread for executing the actions as individual tasks. A worker task first queries for its "observable" timestamp and then appends the actions tagged with this timestamp to the queue. As shown in Figure 1, a transaction worker first increments the global timestamp counter in step one to get the current timestamp to tag its deferred actions. When the transaction begins in step two, it increments the global timestamp counter again. The action thread then processes the queue in order. In particular, the thread checks the timestamp tag of the first item in the queue and compares it to the timestamp of the current oldest transaction. If the oldest transaction's timestamp is larger than the tag, or there are no active transactions, then the action thread pops the head and executes the task. Otherwise, the thread is blocked until this condition is satisfied. If the queue is empty, the action thread waits in the background for a new action to process. For example, in step three of Figure 1, actions tagged with `t1` in the queue get popped and executed because they have tags smaller than the timestamp of the oldest active transaction, `t2`, in the system. After that, the action thread is blocked because the next item in the queue has a tag, `t3`, that is larger than `t2`. The action thread can proceed only when the transaction with timestamp `t2` completes, as shown in steps four and five in Figure 1.

3.3 Ordering Actions

The naïve implementation of this API lacks ordering guarantees for actions from concurrent producers. Even with actions serialized into a single queue, hidden dependencies caused by MVCC and snapshot isolation make certain actions such as deleting the data

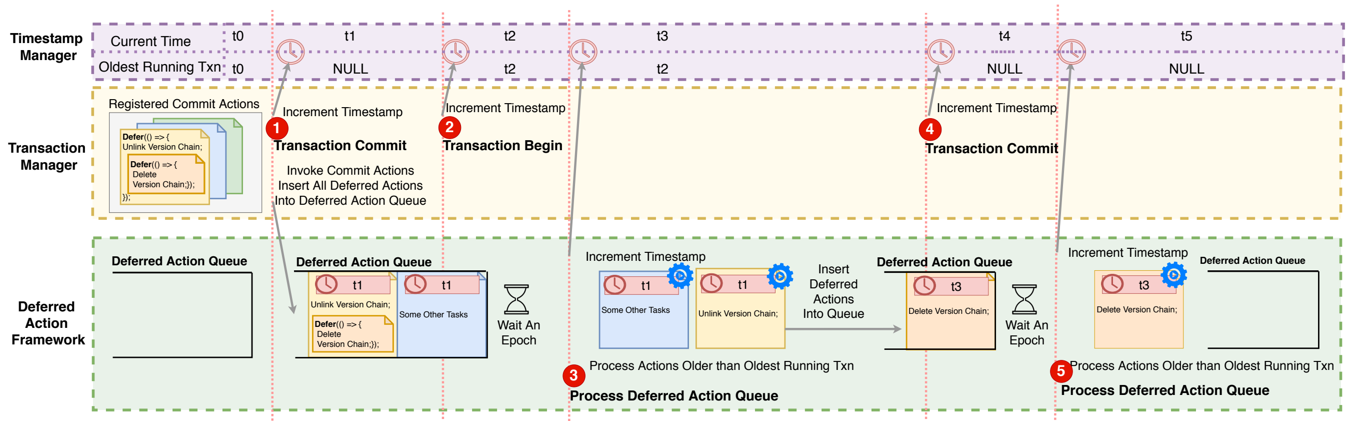

Figure 1: DAF Overview –The DAF integrates with the transaction engine of NoisePage and tags actions with the system timestamp at the time of enqueueing. DAF pops and executes an action if its timestamp is smaller than the oldest running transaction in the system. The shared timestamp between transactions and DAF ensures correct ordering between action processing and transactional access.

structure backing a table problematic. We now present a solution for this without modifying DAF through chaining deferrals.

Chaining deferrals allow us to bootstrap basic guarantees about the ordering of actions within the queue. Consider the following execution under snapshot isolation: transaction T_1 drops a table and commits while transaction T_2 actively inserts into the table and commits after T_1. The action to prune the version chain from T_2 will be processed after any action from T_1, including the deletion of the table data structure DAF must ensure that it processes the delete action after it completes all other actions on the table for memory safety. Observe that because no new transactions will see the table after T_1 commits at time t, any actions referencing the table after t in the defer queue can only come from concurrent transactions such as T_2. We solve this problem with the following chaining of event deferrals: T_1 defers an action that when executed, defers the actual deletion of the table. At the time of the second deferral, all other actions on the table must be already in the queue, and the deletion will be correctly ordered after them, at the tail of the queue. The system can chain deferrals more than once to accommodate more complex ordering requirements, as we will show in Section 4, although in practice, we have not found the need to do so more than twice.

4. OPTIMIZATIONS

The main challenge with general-purpose frameworks like DAF is that they often achieve worse performance than specialized implementations. To overcome this, we now describe optimizations that we developed when integrating DAF into NoisePage.

4.1 Timestamp Caching & Batching Actions

Before processing an action, DAF must know the timestamp of the oldest running transaction. Computing this timestamp per action on-the-fly is expensive, and is a bottleneck on the data structure that the DBMS uses to track the active transaction set [7]. Given the frequency that transactions begin and finish in high-throughput workloads, it is better to optimize the system for transactions at the expense of finding their minimum timestamp.

One way to avoid this problem is to cache the oldest transaction's timestamp. The DAF uses this cached timestamp rather than recalculating it for each action that it has processed. The DBMS periodically calculates the cached timestamp and then continues to use it as long as it is greater than the timestamp of the head of the DAF's queue. This caching concept can be extended to include batching actions since adjacent actions tend to share the same timestamp tag. Thus, you can reduce the number of latch operations on the queue by eagerly dequeuing multiple actions that are ready to process inside the same critical section.

If the DBMS's active transaction set is under heavy contention, one can further improve caching at the cost of a minor latency increase. The DBMS can maintain a second cached timestamp of the oldest running transaction's timestamp. The DBMS then only updates this cached timestamp when it deletes the oldest running transaction. This introduces a small delay in the time from when the DAF could process action (i.e., when it is safe) to when the framework will process it. The benefit, however, is that this caching reduces the contention of the DBMS's active transaction set.

4.2 Multi-Threaded Action Processing

The queue-based implementation discussed in Section 3.2 is inherently single-threaded. This is not a scalable design for modern multi-core systems. We found that using a single thread for DAF is not able to keep up with processing only version chain pruning actions in NoisePage for TPC-C using six execution threads.

The framework can support processing actions on multiple threads by relaxing the ordering guarantee it provides. But adding more consumer threads without additional controls is unsafe. The location of an action in the queue relative to other actions is not enough to guarantee correct ordering. Consider the version chain pruning example. Suppose that one thread is processing an action to remove old versions from a table and then stalls due to a context switch. Then another thread processes an action to delete that same table in response to a DDL statement. When the first thread awakes, it tries to complete the action on a table that no longer exists.

The way to avoid this problem is to process actions inside of transactions. For actions that arise due to schema changes (e.g., the drop table example), the DAF adds a third deferral to ensure that it processes actions in the correct order. As discussed above, all pruning actions (single deferral) are ahead of any action that is double deferred. Thus, if the system pops an action from the queue and executes it within the same transaction, then this guarantees that when the system executes any double-deferred action, all prior actions are either in-progress or completed. Deferring from this point guarantees that all prior actions are done because their associated transactions have completed.

Executing actions in transactions in this manner allows us to associate the length of an action's deferral chain to a specific guarantee in the system:

- **Single-Deferral:** All concurrent transactions have exited.
- **Double-Deferral:** All singly-deferred actions from concurrent transactions have started.
- **Triple-Deferral:** All singly-deferred actions from concurrent transactions have completed.

For our DAF implementation in NoisePage, we use multiple consumers to perform the steps discussed in Section 3.2 within transactions. The consumers commit their transactions whenever they cannot execute the action at the head of the action queue.

4.3 Cooperative Execution

Although multi-threading improves the scalability of DAF, the framework is still susceptible to scalability issues at higher thread counts. Furthermore, if the DBMS uses dedicated threads to process actions, then these threads take away computational resources that could be used to execute transactions and queries. Choosing the right number of DAF threads in this architecture is a trade-off between the system being able to keep up with the background maintenance requirements and losing performance due to context switches.

To avoid this problem, the DAF can employ a *cooperative* execution model where worker threads are also responsible for processing actions [7, 14, 17]. This approach provides two benefits: (1) it creates natural back-pressure on worker threads as delta records accumulate and (2) it improves locality in the memory allocator.

The former helps to prevent a runaway performance situation where the DBMS's garbage collection mechanism cannot keep up with demand [7]. By interspersing actions on the same threads as transactions, the DBMS achieves an equilibrium where it does not produce more actions than it can sustainably execute.

The other benefit is improved locality for the DBMS's memory allocator. Most state-of-the-art allocators, such `jemalloc`, use arenas that it maintains on a per-thread basis. When a thread frees memory, the allocator adds that newly freed memory back to the calling thread's local arena [5]. In this situation, an arena-based allocation scheme is most efficient when the same threads are both allocating and freeing memory, as they do not access a shared memory pool.

5. APPLICATIONS

We next outline the use cases where DAF helped simplify the implementation of NoisePage's components. This discussion is our vision for how other system developers can use DAF to achieve more functionality at lower engineering costs.

5.1 Low-Level Synchronization

The first category encompasses methods for ensuring the correctness and safety of a DBMS's internal physical data structures.

Index Cleaning: Most of the data structures used in DBMSs for table indexes do not natively support multi-versioning [25]. Thus, to use these data structures in an MVCC DBMS, developers either (1) embed version metadata into index keys or (2) maintain version metadata outside of the index. The latter is preferable because the DBMS already does this to identify whether a tuple is visible. With DAF, it is possible to take an existing single-version data structure and integrate it into the DBMS with minimal code to add support for multi-versioning. The high-level idea is to treat any update to an indexed attribute on a tuple as an insert followed by a delete, and then use an action to remove the deleted version when it is no longer visible [20]. The index registers two actions for the updating transaction: a commit action that defers deleting the original key upon commit as well as an abort action that would immediately remove the new index key.

Query Cache Invalidation: DBMSs rely on query plan caching for frequently executed queries and prepared statements to reduce the amount of redundant work. When an application changes the physical layout of a table (e.g., drop column) or changes the indexes on a table, the DBMS may need to re-plan any cached queries. For example, if a cached query plan accesses an index but then the application drops that index, the DBMS needs to invalidate the plan. When the application invokes the query again, the DBMS generates a new plan for it. Concurrent transactions still access the previous query plan if the schema change is non-blocking. With DAF, the transaction that issued the physical layout change only needs to enqueue an action that defers the removal of the old query plan once all the transactions that could access that plan are finished.

Latch-free Block Transformations: Some HTAP DBMSs treat frequently modified (hot) blocks of data and read-mostly (cold) data differently [3, 2, 4]. In NoisePage, transactions modify hot data in-place, and concurrent transactions use version deltas to reconstruct earlier versions. For cold data, NoisePage converts data to a more compact and read-efficient representation in-place [18]. Non-modifying queries are able to read data from cold blocks without checking the version chain and materializing the tuple. During normal operations, the DBMS may need to convert a data block between hot and cold formats multiple times due to changes in the application's access pattern. Because the physical transformation from one format to another is not atomic, the DBMS protects blocks with a shared latch that transactions have to check before accessing them. However, this would lay on the critical path of queries and cause scalability issues. DAF enables the DBMS to perform these layout transformations without such a latch. Instead, the DBMS sets a flag inside of a block's header to indicate that the block is in an intermediate state. It then defers the transformation in an action. Transactions that observe that the intermediate flag is set fallback to materializing tuples when reading, as some threads may still be issuing in-place writes. When the DBMS finally processes the transformation action, all threads are in agreement that the block will not be modified, and thus it safely allows in-place readers.

5.2 Non-Blocking Schema Change

Supporting transactional schema changes are notoriously difficult because they sometimes require the DBMS to rewrite entire tables [19, 23]. In some cases, the schema change is trivial and thus the DBMS does not need to block other transactions (e.g., rename table, drop column). But there are other changes where the DBMS will block other queries until the modification finishes.

With DAF, schema changes become easier to support because the DBMS can reason about transactional physical data structure modifications without special casing or coarse-grained locking. We can extend the drop table example in Section 3.3 to also support indexes and columns; the physical change is concealed via versioned shim functions which are removed once all transactions operate on the same schema definition again. DAF's flexibility enables more interesting possibilities because it can leverage a DBMS's own MVCC semantics to version other optimizations. For example, the DBMS could JIT compile optimized access methods for its physical storage layer, and then use actions to maintain a catalog of these methods. The DBMS could also load new indexes or storage engines into its address space at runtime without having to restart. Again, DAF makes this possible because the deallocation mechanisms for these components are decentralized, which means that they are not dependent on hard-coded logic in the DBMS's GC routines.

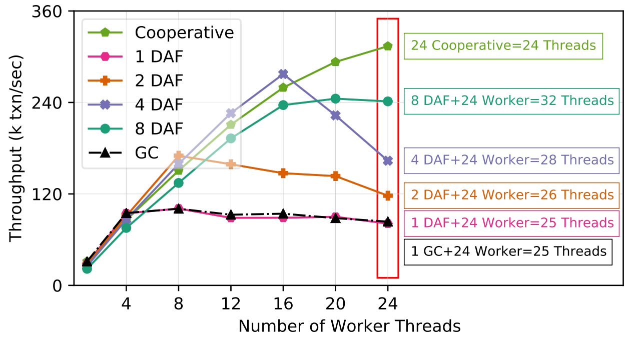

Figure 2: TPC-C Performance – Throughput comparisons when varying the number of worker threads using (1) cooperative DAF threads, (2) a single dedicated GC thread, and (3) dedicated action processing threads.

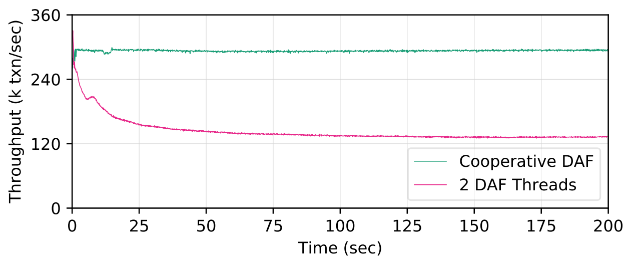

Figure 3: Cooperative vs. Dedicated DAF Threads (Throughput) – Comparison for NoisePage with a total of 20 threads, using either (1) cooperative action processing or (2) two dedicated action processing threads.

6. PRELIMINARY RESULTS

We now present our evaluation of DAF in NoisePage. Our goal is to demonstrate the transactional performance for our DAF implementation and showcase its support for easy extension and instrumentation. We perform all experiments on Amazon EC2 `r5.metal` instance: Intel Xeon Platinum 8259CL CPU (24× cores, HT disabled) with 768 GB of memory.

We use transactional GC as our sample use-case for these experiments. We compare our DAF-based implementation against an earlier version of NoisePage with a hand-coded GC similar to the original HyPer [20]. We use the TPC-C workload with one warehouse per worker thread and report the total number of transactions processed. We pre-compute all of the transaction parameters and execute each transaction as a stored procedure. The DBMS run 200 seconds per trial to ensure the system reaches steady-state throughput, and we record performance measurements using NoisePage's internal metric logging framework.

The graph in Figure 2 shows NoisePage's throughput when increasing the number of worker threads. DAF is able to scale across multiple cores when using (1) cooperative or (2) dedicated thread action processing. The latter outperforms the cooperative configuration below 16 worker threads, but we see a drop in performance when the total number of threads in the system (dedicated + worker) saturates the number of physical cores. These results show that the dedicated thread configuration fails to scale when exceeding four worker threads per DAF thread.

To better understand this performance degradation for higher thread counts, we measure the DBMS's throughput continuously during the benchmark. Figure 3 shows the sustained throughput of two runs with 20 threads on two configurations: (1) cooperative and (2) two dedicated DAF threads. The dedicated thread configuration initially starts with approximately the same throughput as cooperative, but then its performance drops by half within the first 30 seconds of execution. To explain this pattern, we plot average

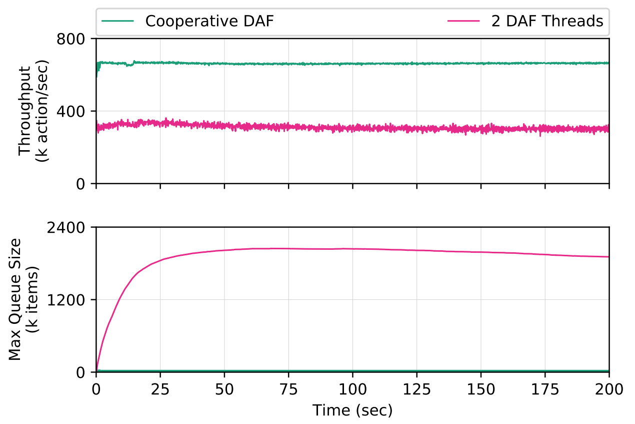

Figure 4: Cooperative vs. Dedicated DAF Threads (Metrics) – DAF's internal measurements from the experiment in Figure 3 for (1) action processing throughput and (2) max action queue size.

the action processing rate and queue size over time in Figure 4. With cooperative threading, we observe both a steady throughput on actions processed as well as a negligible actions queue size. In contrast, the two DAF thread configuration shows a lower throughput of actions. This throughput is not sufficient to keep up with the maintenance demand of 20 workers, and thus the size of the action queue increases by several orders of magnitude. As this happens, tuples' version chains become longer, indexes become larger, and transactional throughput lowers, until the system eventually reaches a steady state. This steady state is undesirable as it corresponds to a lower throughput, and several seconds of average latency for actions versus sub-millisecond latency with the cooperative configuration.

7. RELATED WORK

To the best of our knowledge, there is no previous work on building a general-purpose framework to maintain internal physical data structures of a DBMS with transaction timestamps. Our work is inspired by and builds on advancements in MVCC and epoch-based GC for in-memory DBMSs.

Garbage Collection in MVCC: There are two representative approaches to GC in MVCC systems. Microsoft Hekaton uses a cooperative approach where actively running transactions are also responsible for version-chain pruning during query processing [9]. Transactions refer to the "high watermark" (i.e., the start timestamp of the oldest active transaction) to identify obsolete versions. SAP HANA periodically triggers a GC background thread using the same watermarks [15]. HANA also uses an interval-based approach where the DBMS prunes unused versions in the middle of the chain (as opposed to only the head of chain as in Hekaton). HyPer's Steam improves techniques from Hekaton and HANA: it prunes both the head of version chains as well as the middle of chains by piggy-backing the GC tasks on transaction processing [7]. The same methods that the DBMS uses to identify obsolete versions (e.g., high watermark, interval-based) are orthogonal to DAF. DAF's support for cooperative processing allows it to have a higher GC frequency compared to background vacuuming. Moreover, DAF is a general framework that can do more than GC: as described in Section 5, version-chain pruning is one of the applications that DAF supports.

Epoch Protection: One can also consider DAF to be an epoch protection framework, which is widely used in multi-core DBMSs. FASTER's epoch protection framework exposes an API similar to DAF for threads to register arbitrary actions for later execution [8]. FASTER is a non-transactional embedded key-value store, and its epoch framework maintains its own counter that is cooperatively advanced by user threads. These threads must explicitly refresh the epoch framework and process actions periodically to guarantee progress. FASTER also offers no ordering guarantees between actions registered to the same epoch, whereas NoisePage can accommodate this with repeated deferrals. Although not multi-versioned, Silo's concurrency control protocol relies on epochs [26]. The system maintains a global epoch counter that increments periodically, and transactions from larger epochs never depend on smaller epochs. The Bw-tree [17, 27] is a latch-free data structure from Hekaton that relies on a similar epoch-based GC scheme like Silo.

Memory Management: A DBMS's memory allocator also affects a DBMS's performance in a multi-core environment [5, 13, 16]. The allocator will affect the DBMS's resident set size, query latency, and query throughput of a DBMS [10]. Although not thoroughly studied in the context of a DBMS, these allocators also cause performance variations depending on whether the threads allocating memory are also the same ones freeing it [5]. Because DAF turns GC into a thread-independent, parallelizable task, it is worth exploring the interaction of GC parameters with allocators [7, 16].

Since DAF introduces transactional semantics to data structure maintenance, it has some similarities with software transactional memory (STM) [11]. STM instruments program instructions to provide transactional semantics to memory reads and writes. In contrast, DAF is not by itself transactional, but integrates into a transactional engine to complement its capabilities. DAF also operates at a higher abstraction level than STM, operating on program-level maintenance tasks, as opposed to instruction-level.

8. FUTURE WORK

We foresee more optimizations beyond those in Section 4 that could improve the DAF's scalability. We now discuss two of these as potential research directions.

Coalescing Deferrals: Another way to reduce contention on the action queue latch is for threads to coalesce their *observed* deferrals into a single action. This extra processing step is likely to be a substantial performance improvement in the common case. But it requires a protection mechanism to ensure that long-running transactions do not inadvertently create a single, long-running action. Such transactions would create a second pause in the framework since actions must be executed inside of transactions. The DBMS could minimize this risk by limiting the number of actions that threads are allowed to combine together.

Long-Running Transactions: The most onerous short-coming of our current implementation of DAF is that it assumes read-write transactions are short-lived. Action processing is halted if the oldest running transaction in the system does not exit. This is similar to the impact of long-running transactions in [20], or the impact of a thread does not refresh its epoch in [8]. It is possible to use techniques outlined in [15] and [7] to ensure progress of the rest of the system, although this invariably leads to additional complexity in the API and implementation of DAF.

9. CONCLUSION

We presented the Deferred Action Framework for unifying the life cycle of transactional logical database objects and physical data structures in a DBMS. This framework integrates with existing an DBMSs to leverage MVCC semantics for the system's internal maintenance tasks. Our evaluation DAF in the NoisePage DBMS shows that achieves similar or better performance for version chain and index clean-up while keeping the corresponding code straightforward and modular. We also presented other maintenance scenarios that DAF could support, such as non-blocking schema changes.

10. REFERENCES

[1] NoisePage. https://noise.page.

[2] I. Alagiannis, S. Idreos, and A. Ailamaki. H2o: A hands-free adaptive store. SIGMOD, pages 1103–1114, 2014.

[3] J. Arulraj et al. Bridging the archipelago between row-stores and column-stores for hybrid workloads. SIGMOD, pages 583–598, 2016.

[4] M. Athanassoulis, K. S. Bøgh, and S. Idreos. Optimal column layout for hybrid workloads. *Proc. VLDB Endow.*, 12(13):2393–2407, 2019.

[5] E. D. Berger et al. Hoard: A scalable memory allocator for multithreaded applications. ASPLOS, pages 117–128, 2000.

[6] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[7] J. Böttcher, V. Leis, T. Neumann, and A. Kemper. Scalable garbage collection for in-memory mvcc systems. *Proc. VLDB Endow.*, 13(2):128–141, Oct. 2019.

[8] B. Chandramouli et al. Faster: A concurrent key-value store with in-place updates. SIGMOD, pages 275–290, 2018.

[9] C. Diaconu et al. Hekaton: Sql server's memory-optimized oltp engine. SIGMOD, pages 1243–1254, 2013.

[10] D. Durner, V. Leis, and T. Neumann. On the impact of memory allocation on high-performance query processing. In *DaMoN*, pages 21:1–21:3, 2019.

[11] N. Herman et al. Type-aware transactions for faster concurrent code. EuroSys'16.

[12] K. Kim et al. Ermia: Fast memory-optimized database system for heterogeneous workloads. SIGMOD, pages 1675–1687, 2016.

[13] P.-Å. Larson and M. Krishnan. Memory allocation for long-running server applications. ISMM, pages 176–185, 1998.

[14] P.-r. Larson et al. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309.

[15] J. Lee et al. Hybrid garbage collection for multi-version concurrency control in sap hana. SIGMOD, pages 1307–1318, 2016.

[16] D. Leijen, B. Zorn, and L. de Moura. Mimalloc: Free list sharding in action. Technical Report MSR-TR-2019-18, Microsoft, June 2019.

[17] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. ICDE, pages 302–313, 2013.

[18] T. Li et al. Mainlining databases: Supporting fast transactional workloads on universal columnar data file formats, 2020.

[19] J. Løland and S.-O. Hvasshovd. Online, non-blocking relational schema changes. In *EDBT*, pages 405–422, 2006.

[20] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. SIGMOD, pages 677–689, 2015.

[21] A. Pavlo et al. Self-driving database management systems. CIDR'17.

[22] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *Proc. VLDB Endow.*, 5(12):1850–1861, Aug. 2012.

[23] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek. Online, asynchronous schema change in f1. *Proc. VLDB Endow.*, 6(11):1045–1056, Aug. 2013.

[24] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in sap hana database: The end of a column store myth. SIGMOD '12, pages 731–742, 2012.

[25] Y. Sun et al. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proc. VLDB Endow.*, 13:221–225, 2019.

[26] S. Tu et al. Speedy transactions in multicore in-memory databases. SOSP, pages 18–32, 2013.

[27] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a bw-tree takes more than just buzz words. SIGMOD, pages 473–488, 2018.

[28] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, Mar. 2017.