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). Still, 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 engineering efforts in coordinating 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.

1. INTRODUCTION

Race conditions and transaction interleavings within an MVCC DBMS remain implementation hurdles despite insights from decades of development [10, 25, 23, 31, 21]. This difficulty partly arises from 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 [28]. For example, an MVCC version chain entry is obsolete when it is no longer visible to any active transactions in the system. The DBMS's garbage collector can then look up the oldest running transaction in the system and safely remove the entries created before that transaction starts [16, 8].

In this paper, we generalize this idea into a modular component, called the **Deferred Action Framework** (DAF), that processes maintenance tasks safely and scalably. 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. It provides epoch protection to transactions and maintenance tasks without requiring a separate mechanism for refreshing and advancing epochs. Unlike other epoch-based protection implementations [28], 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).

DAF helps us reason about more complex transaction interleavings in databases with evolving schemas and serves as a basis for supporting non-blocking schema changes. Because DAF decouples action processing from action generation, it gives system developers flexibility to adjust action processing strategies dynamically. 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 these tasks' implementation complexity while offering competitive performance compared to hand-optimized alternatives. Additionally, we find DAF to be a natural central point for runtime metrics collection in our system. We use the size of DAF's internal action queue to detect when NoisePage is behind in some maintenance tasks, such as version chain pruning, and raise the issue to the rest of the system for handling.

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 [7]. Under this scheme, readers can access 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. For example, systems need to store and differentiate multiple versions, maintain them until no transaction can access them, and discard them to free up storage space afterward. In this section, we provide an overview of these challenges and a brief survey of existing solutions. We also describe NoisePage's MVCC implementation to help ground the discussion about integrating DAF into a DBMS.

2.1 The NoisePage System

NoisePage is a relational HTAP DBMS developed at Carnegie Mellon University [1]. The DBMS stores all tuples in a PAX-style in-memory columnar format based on Apache Arrow [19]. NoisePage uses HyPer-style optimistic concurrency control with MVCC [21, 31], and all transactions execute under snapshot isolation. Under this scheme, each transaction in the system obtains two unique timestamps — t_{start} and t_{end}. On a high-level, transactions write entries at t_{end} and read entries at t_{start}. Transactions only read values that are committed before they start and abort when encountering write-write conflicts.

We implement the concurrency control scheme newest-to-oldest version chains of delta entries. Each version chain entry contains t_{end} of the writer. Uncommitted transactions write negative t_{end} to signal that its writes are not yet visible and update their delta records with the correct t_{end} only after they commit. When updating, a transaction first copies the before-image of updated tuple attributes to a new version chain entry, atomically installs it onto the version chain, and then proceeds to modify the tuple in-place. For updates to indexed attributes, NoisePage models them as a delete followed by an insert. Even though NoisePage implements a well-studied concurrency control scheme, it comes with much complexity due to concurrent versions' presence in all layers of the system. Consequently, like other systems of this type, NoisePage requires an involved garbage collection scheme and other maintenance to function properly, which we will discuss next.

2.2 Data Structure Maintenance in MVCC

Multi-versioning permeates an MVCC DBMS's internal data structures. For example, DBMS indexes must handle multiple references pointing to the same logical tuple when its physical versions differ in indexed attributes [31]. The system needs to accommodate multiple versions of the schema co-existing to support non-blocking schema changes [20], which leads to multiple entries in the system's query plan cache for the same operation. Additionally, data structure maintenance happens concurrently with user transactions and must coordinate their access for memory safety and semantic correctness. Such coordination must be scalable not to affect user transaction performance.

One might attempt to yield the most efficient schedule for maintenance tasks by explicit dependency tracking, perhaps using a dependency graph like TensorFlow [2]. However, unlike TensorFlow tasks, which have pre-defined static dependencies, DBMS dependencies dynamically evolve as users issue new transactions, making this unrealistic in a performance-critical setting. Systems resort to coarse-grained epoch-based protection instead [28, 9, 13], which prevents the maintenance tasks from accessing memory still accessible by running transactions. Under this scheme, transactions protect against concurrent maintenance tasks with a monotonically increasing global counter or epoch and an epoch table of all running transactions. Worker threads initiate the protection by registering their thread-local epochs in the global epoch table. They then deregister after they no longer require protection. The epoch steadily advances, so eventually, an epoch has no registered transactions in the epoch table and becomes unprotected. The system can safely process the maintenance tasks associated with that epoch 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, and developers must maintain the epoch counter explicitly during query processing.

3. FRAMEWORK OVERVIEW

This section presents an overview of DAF, its logical model, and its programming interface. Central to DAF is the concept of *actions*, which 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.

The DAF API exposes a single function to the rest of the system: $\mathit{defer}\ (\mathit{action})$. This function takes in an action as a lambda function that captures required references, then tags the action with the current timestamp. DAF guarantees to invoke the given action only after there are no transactions in the system with a start timestamp *smaller* than the tagged timestamp.

3.1 System Characteristics

DAF requires the DBMS concurrency control algorithm to satisfy specific properties. In particular, the DBMS must 1) correctly order all transactions by their begin timestamp, 2) track the oldest active transaction by begin timestamp, and 3) provide an "observable" timestamp for each transaction, at which point all concurrent and future transactions are guaranteed to see their logical effects. Although the discussion below is specific to our implementation of DAF in NoisePage, which has a single global counter for timestamps, the framework functions correctly as long as the concurrency control scheme meets the criteria mentioned above.

For threads in our system, the unit of work 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* that execute individual transactions in the system and *action tasks* that are maintenance routines to clean up and release resources no longer in use.

3.2 Implementation

Our DAF contains a queue that holds actions tagged with an "observable" timestamp. When a worker task, or action, is generated during transaction executions, the worker thread first queries the action's "observable" timestamp from the system's global timestamp counter. Then the worker thread appends the action tagged with this timestamp to the queue. The global timestamp counter increment as the system begins and ends a transaction. The system keeps track of the timestamps of running transactions, and DAF uses these timestamps and timestamp tags to decide whether it should execute an action. DAF effectively uses the oldest running transaction's

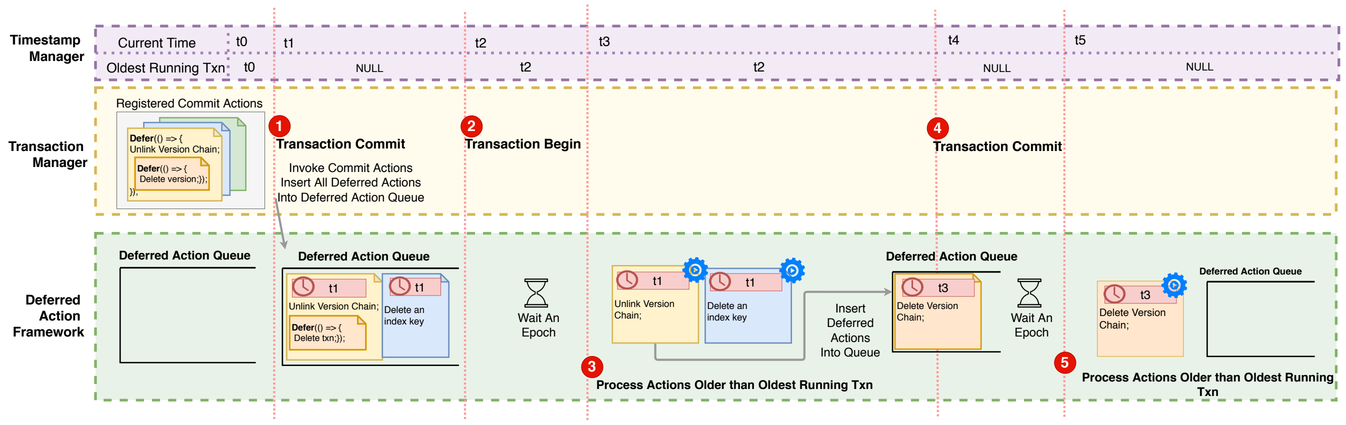


Figure 1: DAF Overview –DAF integrates with the transaction engine of NoisePage and tags actions with the system timestamp at the time of enqueueing. DAF pops and executes an action if its timestamp is smaller than the oldest running transaction in the system. The shared timestamp between transactions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

start timestamp as an epoch to protect actions that are unsafe to execute without having to maintain a separate data structure. Figure 1 presents how DAF interacts with the system's transaction manager and timestamp manager with an example. In the simplest configuration, DAF utilizes a single action queue and a dedicated thread for executing the actions as individual tasks.

As shown in Figure 1, a transaction worker first increments the global timestamp counter in step (1) to get the current timestamp to tag its deferred actions. When the transaction begins in step (2), 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 new actions to process. For example, in step (3) of Figure 1, actions tagged with `t1` in the queue get popped and executed because they have tags smaller than the oldest active transaction's timestamp, `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 (4) and (5) in Figure 1.

3.3 Ordering Actions

Even with actions serialized into a single queue, hidden dependencies between actions caused by MVCC and snapshot isolation make certain actions such as deleting the data structure backing a table problematic. 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. T_1 generates an action that deallocates the table, and T_2 generates an action to prune the version chain in the same table. If the action from T_1 is inserted before the action from T_2, DAF will delete the table before pruning the version chain. Therefore, the system will access a memory location that has already been deallocated while executing the action from T_2. We present a solution for this without modifying DAF through chaining deferrals. Chaining deferral can be demonstrated with DAF API as $defer\ (defer\ (...defer\ (action)\ ...))$., where multiple "deferring"s wrap over a single action.

Chaining deferrals allows DAF to bootstrap basic guarantees about the ordering of actions within the queue. We now revisit the previous problem with this concept. The action to prune the version chain from T_2 must be processed after deleting the table data structure. DAF needs to ensure that it processes the delete action after completing 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. When DAF pops this action from the queue and executes, the action will defer another action: the actual deletion of the table. At the time of the second deferral, all other actions on the table are in the queue. Thus the deletion will be correctly ordered after them, at the tail of the queue. queue. However, in the existence of multiple consumers, chaining a deferral once can not guarantee correct execution ordering. The system can chain deferrals more times to accommodate more complex ordering requirements, as we will show in Section 4.1. In practice, we have not found the need to chain a deferral more than two times.

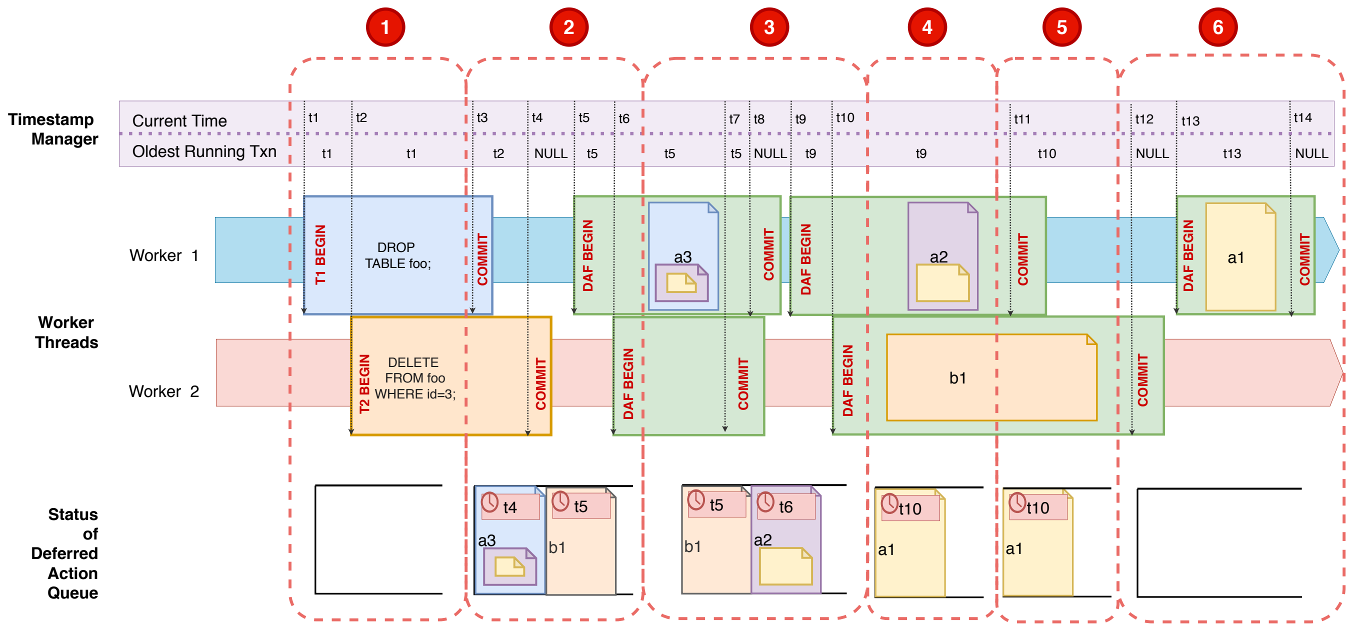
4. OPTIMIZATIONS

The main challenge with general-purpose frameworks like DAF is that they are often less performant than specialized implementations. We now describe optimizations that we developed when integrating DAF into NoisePage that address this issue.

4.1 Multi-Threaded Action Processing

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

Concurrently processing actions from the queue with multiple threads weakens the ordering guarantee of DAF, making it generally unsafe without special handling. Specifically, previously introduced mechanisms only ensure the relative ordering of actions in the queue, which corresponds to when to start actions. Unlike single-threaded DAF, where the singular thread cannot start new actions before finishing earlier ones, multiple threads can complete actions out-of-order. Subsequently, the thread that finishes first will start processing later actions before earlier ones have finished. Again 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 will complete the action on a table that no longer exists, which leads to memory errors. We



1
2
3
4
5
6
Timestamp Manager
Current Time
Oldest Running Txn
t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14
t1 t1 t2 NULL t5 t5 t5 NULL t9 t9 t10 NULL t13 NULL
Worker Threads
Worker 1
Worker 2
T1 BEGIN
DROP TABLE foo;
COMMIT
T2 BEGIN
DELETE FROM foo WHERE id=3;
DAF BEGIN
a3
a2
a1
b1
Status of Deferred Action Queue
t4
t5
t6
t10

Figure 2: Triple Deferral –In the presence of multiple consumers to the action queue, chaining a deferral two times (triple deferral) can guarantee that the system executes the action only after executing all singly deferred actions from concurrent transactions. By processing actions inside a transaction, the thread having the oldest running transaction effectively set the global "fence" that marks the processable actions for all consumers.

make two adjustments to DAF to address this problem: running actions in transactions and chaining an additional deferral for actions with implicit dependencies, such as DDL changes.

The first adjustment is to have DAF threads process actions inside transactions. On a high level, this adds the same epoch protection for user transactions to actions, which prevents concurrent DAF threads from advancing too far ahead of each other. Consider a case where there are no active user transactions in the system; by DAF's semantics, all actions are safe to process. This extends to actions generated by a double-deferred action, making it possible for another processing thread to prematurely execute the double-deferred action. If the DAF thread instead starts a transaction periodically, other threads cannot process such generated actions before the transaction is finished. Note here that it is unnecessary to start a transaction for each action, as long as an action is run within the duration of some transaction's protection. In practice, we reuse the same transaction for multiple actions for efficiency.

The second adjustment is adding a third deferral to actions that arise due to schema changes (e.g., the drop table example). As discussed above, DAF guarantees that all pruning actions (single deferral) are started before any concurrent double-deferred actions. Thus, if the system pops an action from the queue and executes it within the same transaction, all singly-deferred actions from concurrent transactions are guaranteed to be either in-progress or completed when the system executes any double-deferred action. That said, this is insufficient, as in-progress actions can be arbitrarily delayed and perform unsafe operations, such as traversing the version chain, after the double-deferred action is processed. Deferring from this point, though, guarantees that all prior actions have been completed. Their associated transactions must have finished for the epoch to have advanced to the deferred execution time.

To provide a more concrete example of how this guarantees correctness, we walkthrough the worst-case ordering of this action in Figure 2. For the sake of simplicity, we assume that the system's worker threads handle both transactions and DAF actions but leave the full discussion of this change to Section 4.3.

1. The example starts with the two worker threads executing our concurrent DDL and DML transactions as previously described and an empty DAF queue.

2. In the worst-case setting, the DDL transaction commits ($t3$) before the DML transaction, which places the deferral chain of the table into the DAF action queue before the action to unlink the delete's undo records from the table. Note that in our implementation, a worker thread can insert an action to the action queue during or after the transaction commits, as long as the insertion happens before the next transaction starts in the same thread. The opening of a transaction at $t5$ in Worker 1 and transaction at $t6$ in Worker 2 shows the first important change required for safe, concurrent processing of actions. Without this step, multiple deferrals would have no effect as worker threads could always process the next action when no normal transactions are active. This could result in Worker 1 processing a's entire deferral chain while Worker 2 is executing b. In contrast, by checking out transactions, $a2$ cannot be started until all concurrent transactions with $a3$ have been completed.

3. Continuing our worst-case scenario, Worker 1 completes $a3$ and inserts $a2$ at the back of the queue. Since the current oldest running transaction (Worker 1's DAF transaction) is not greater than the timestamp associate with $b1$, Worker 1 commits and starts again. Transaction at Worker 2 ($t6$) commits without being able to process any actions.

4. During this window, Worker 1 fully executes $a2$ while Worker 2 starts $b1$ stalling on some step of that action. It is essential to observe that at the moment $a2$ commits, all singly-deferred actions are either complete or running in a separate transaction of the other worker thread. It also highlights that double-deferrals are not sufficient for DDL safety as $a2$ would have been the actual table deletion in that case and could have occurred before $b1$ dereferenced the table for unlinking.

5. Because the timestamp associated with $a1$ is $t10$, Worker 1 commits and then stalls because even after exiting, the oldest running transaction is only $t10$. It is the critical moment where triple-deferrals achieve the required ordering guarantee because NoisePage's epoch system for transactions now also serves as a guard on DAF preventing early execution of deferred actions.

6. Once Worker 2 finishes $b1$, it is also unable to execute $a1$ because of the oldest running transaction. Therefore it commits, allowing Worker 1 to resume and execute $a1$ inside of a transaction – deleting the table only after all singly-deferred references are guaranteed to complete.

For DAF queue with multiple concurrent consumers, executing actions inside transactions with up to triple deferral 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 before a singly-deferred action can start.
- **Double-Deferral:** All singly-deferred actions from concurrent transactions have started.
- **Triple-Deferral:** All singly-deferred actions from concurrent transactions have been completed.

4.2 Timestamp Caching & Batching Actions

Before processing an action, DAF must know the timestamp of the oldest running transaction. Computing this timestamp per action is expensive and can create additional contention on timestamp generation for transactions [8], which is undesirable for high-throughput workloads. In such cases, it is desirable to trade-off some accuracy of the minimum timestamp computation for better transactional performance. Caching is one such trade-off. DAF uses two levels of caching for the oldest transaction's timestamp for use across multiple actions.

The first cache is a pre-computed value stored separately from the set of current running transaction timestamps. This pre-computed value is updated whenever removing the current oldest transaction from the running transaction set, and DAF only reads from this pre-computed value. This has significant performance benefits: it removes DAF threads as a source of contention on the current running transaction set and ensures that computation of the oldest running transaction is only ever done once per epoch. It is also worth noting that this cache can easily be modified to allow further tuning by making the policy for when to update the cached value as a runtime parameter.

The second cache is per DAF thread and is simply a local copy of the first-level cache which is only updated when the timestamp for the action at the head of the action queue exceeds its locally cached version. In this situation, the thread refreshes its local cache from the global cache and either tries again if it changed or commits its DAF transaction if it did not.

These caches can only delay the execution of actions in the aggregate, and cannot lead to early execution of an action or an action being removed from the queue outside of a transaction and therefore do not affect the correctness or ordering guarantees from Section 3.2 and Section 4.1.

We can further extend this caching concept to include batching actions since adjacent actions tend to share the same timestamp tag. Since multiple threads are concurrently accessing the action queue, they compete for the queue latch each time an action is pushed into or popped out of the action queue. Thus, we can reduce the number of latch operations on the queue by eagerly dequeuing multiple actions inside a single trip through the critical section.

4.3 Cooperative Execution

Although multi-threading improves DAF's scalability, the framework is still susceptible to scalability issues at higher thread counts. The number of actions increases proportionally with the number of concurrent worker threads. When DAF fails to keep up with the action generated, actions will accumulate and affect the system's throughput. For example, when actions of cleaning up deleted index keys accumulate, each later transaction sees a larger index and takes more time to traverse the index. Therefore, more dedicated DAF threads are needed in the case of higher worker thread count, especially when DAF operates on a workload that generates many maintenance tasks. However, this way, we add more contention to the action queue on top of the contention introduced when scaling up the number of worker threads. Additionally, the static allocation of dedicated threads cannot react to workload change due to a lack of back-pressure.

To avoid this problem, DAF can employ a *cooperative* execution model where worker threads are also responsible for processing actions [8, 15, 18]. 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 prevent a runaway performance situation where the DBMS's garbage collection mechanism cannot keep up with demand [8]. 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 [6]. In this situation, an arena-based allocation scheme is the most efficient when the same threads are both allocating and freeing memory, as they do not access a shared memory pool. Additionally, cooperative execution eliminates the need to schedule and manage dedicated DAF threads separately.

5. APPLICATIONS

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

5.1 Low-Level Synchronization

We now discuss how we use DAF to ensure the correctness and safety of NoisePage's internal physical data structures in addition to version chains.

Index Cleaning: Most of the data structures used in DBMSs for table indexes do not natively support multi-versioning [26]. 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, we can 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. The index registers two actions for the updating transaction: a commit action that defers deleting the original key upon commit and 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 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 old query plan's removal 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 [4, 3, 5]. 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 [19]. Non-modifying queries can 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. The system must prevent in-place readers and in-place writers from operating on the same block concurrently during this process. DAF enables the DBMS to perform these layout transformations without excessive synchronization (e.g., a block-level latch) between readers and writers. Instead, the DBMS sets a flag inside a block's header to indicate that it is in an intermediate state. It then defers the transformation in an action. Transactions that observe the intermediate flag 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 have agreed not to modify the block, 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 [20, 24]. 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 queries until the modification finishes.

With DAF, schema changes become easier to support because the DBMS can layer transactional semantics on top of physical data structure modifications without special casing or coarse-grained locking. In particular, we can extend support to more complicated DDL queries, such as modifying a column's type by creating specialized functions for reading and modifying data in the tables. The DDL transaction can then create and start using new versions of the functions that can translate data stored in the old format to the new schema while concurrent transactions continue to use the existing function. The pending physical change is concealed by the catalog for concurrent transactions and the new functions for future transactions. Migrating old data to the new format can then be a flexible policy decision for the larger DBMS, with the end of the migration triggering another transactional update to the functions in the catalog. DAF's flexibility becomes apparent for successive schema changes that cause data to be stored in three or more formats simultaneously. In this situation, using a non-versioned data structure such as a map or array greatly simplifies the functions

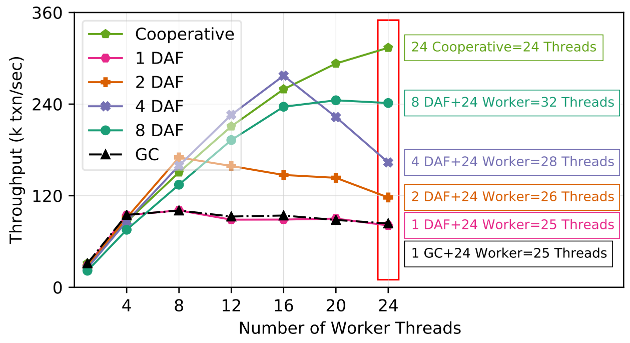


Figure 3: TPC-C Performance – Throughput comparisons when varying the number of worker threads using (1) cooperative DAF threads, (2) a single dedicated GC thread, and (3) dedicated action processing threads.

for accessing the data but would generally pose significant difficulties for a transactional implementation. However, with deferred actions, updates to this data structure can easily be delayed to the right moments in time to enable transactional semantics despite the non-versioned implementation underneath.

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 load new index implementations 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.

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 [21]. We use the TPC-C [27] 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 runs 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 3 shows NoisePage's throughput when increasing the number of worker threads. DAF can scale across multiple cores when using (1) cooperative or (2) dedicated thread action processing. The latter outperforms the cooperative configuration below 16 worker threads. Still, 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 continuously measure the DBMS throughput during the benchmark. Figure 4 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

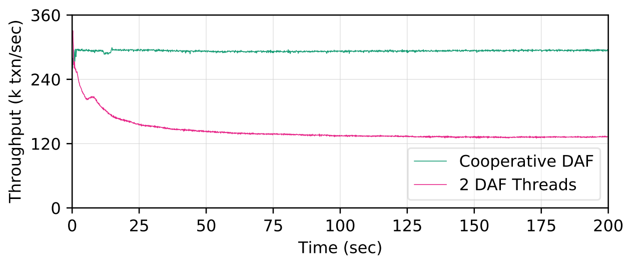


Figure 4: Cooperative vs. Dedicated DAF Threads (Throughput) – Comparison for NoisePage with a total of 20 threads, using either (1) cooperative action processing or (2) two dedicated action processing threads.

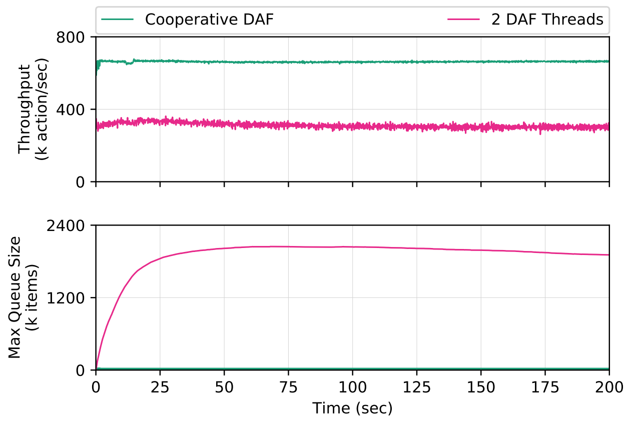


Figure 5: Cooperative vs. Dedicated DAF Threads (Metrics) – DAF's internal measurements from the experiment in Figure 4 for (1) action processing throughput and (2) max action queue size.

cooperative, but then its performance drops by half within the first 30 seconds of execution. To explain this pattern, we plot the average action processing rate and queue size over time in Figure 5. We observe both steady throughput on actions processed and a negligible actions queue size with cooperative threading. 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 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 three 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 [10]. 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 [16]. HANA also uses an interval-based approach where the DBMS prunes unused versions in the middle of the chain (instead of 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 and the middle of chains by piggy-backing the GC tasks on transaction processing [8]. The DBMS uses the same methods to identify obsolete versions (e.g., high watermark, interval-based) 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 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 [9]. FASTER is a non-transactional embedded key-value store, and its epoch framework maintains a 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 [28]. The system maintains a global epoch counter that increments periodically, and transactions from larger epochs never depend on smaller epochs. The Bw-tree [18, 30] 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 [6, 14, 17]. The allocator will affect the DBMS's resident set size, query latency, and query throughput of a DBMS [11]. 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 [6]. Because DAF turns GC into a thread-independent, parallelizable task, it is worth exploring GC parameters with allocators [8, 17].

Since DAF introduces transactional semantics to data structure maintenance, it has some similarities with software transactional memory (STM) [12]. 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 instead of instruction-level.

8. FUTURE WORK

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

Multiple Action Queues: Assigning multiple concurrent producers and consumers to a single action queue can introduce contention and present a scalability bottleneck with large thread counts. One way to reduce contention is to have multiple action queues. By distributing concurrent operations to multiple queues, we can improve the performance of both DAF and the system in general. The system could easily incorporate multiple action queues into the multi-threaded or cooperative framework designs. To guarantee the correct action processing order, any consumer of actions must traverse the set of queues and process as many actions as possible before committing. The strategy for inserting the actions into queues is less strict and should optimize for reducing contention.

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 considerable 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.

Unified Task Queue: In our current implementation, DAF is a stand-alone component with its queue. By combining the queue for normal transactional tasks and the queue of deferred actions, we can have fewer data structures to maintain and synchronize. Workers threads can execute both types of tasks, and we have discussed the correctness of this strategy in Section 4.3 on cooperative execution. With one unified task queuing component in the DBMS, the system can leverage the existing load-balancing strategies on execution and maintenance tasks.

Long-Running Transactions: The most onerous shortcoming of our current implementation of DAF is that it assumes user transactions are short-lived. Action processing will halt if the oldest running transaction in the system does not finish. It will result in a similar impact of long-running transactions in [21], or the effect of a thread does not refresh its epoch in [9]. It is possible to use techniques outlined in [16] and [8] to ensure the rest of the system's progress. However, this invariably leads to additional complexity in the API and implementation of DAF.

Other Maintenance Tasks: There are several other maintenance tasks in a DBMS that could benefit from centralized coordination. For example, lazy materialized view maintenance [33] defers materialized view updates until spare cycles are available in the system. While there is less concern with memory safety in such use cases, DAF can simplify these tasks by providing a unified background task scheduling interface. DAF can perceivably associate a scheduling priority with different types of tasks and choose to chain additional deferrals for low-priority tasks when the system is under load.

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 DBMSs to leverage MVCC semantics for the system's internal maintenance tasks. It introduces extra tuning knobs, such as cooperative or multi-threaded strategy, number of threads, and batch size. How to automatically tunning these parameters is beyond this paper's scope, as there are existing tools to achieve this under the research of self-driving databases [22, 29, 32]. Our evaluation of DAF in NoisePage shows 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.

Acknowledgements

This work was supported (in part) by the National Science Foundation (IIS-1846158, IIS-1718582, SPX-1822933), Google Research Grants, and the Alfred P. Sloan Research Fellowship program. TKBM.

10. REFERENCES

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

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283. USENIX Association, 2016.

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

[4] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. SIGMOD, pages 583–598, 2016.

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

[6] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. ASPLOS, pages 117–128, 2000.

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

[8] 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.

[9] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. Faster: A concurrent key-value store with in-place updates. SIGMOD, pages 275–290, 2018.

[10] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. SIGMOD, pages 1243–1254, 2013.

[11] 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.

[12] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. EuroSys, 2016.

[13] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. SIGMOD, pages 1675–1687, 2016.

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

[15] P.-r. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, 2011.

[16] J. Lee, H. Shin, C. G. Park, S. Ko, J. Noh, Y. Chuh, W. Stephan, and W.-S. Han. Hybrid garbage collection for multi-version concurrency control in sap hana. SIGMOD, pages 1307–1318, 2016.

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

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

[19] T. Li, M. Butrovich, A. Ngom, W. S. Lim, W. McKinney, and A. Pavlo. Mainlining databases: Supporting fast transactional

workloads on universal columnar data file formats. *Proc. VLDB Endow.*, 14(4):534–546, 2021.

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

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

[22] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR*, 2017.

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

[24] 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.

[25] 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.

[26] Y. Sun, G. E. Blelloch, W. S. Lim, and A. Pavlo. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proc. VLDB Endow.*, 13:221–225, October 2019.

[27] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`, June 2007.

[28] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. SOSP, pages 18–32, 2013.

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

[30] 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.

[31] 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.

[32] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. SIGMOD '19, page 415–432, 2019.

[33] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 231–242. VLDB Endowment, 2007.