FineLine: log-structured transactional storage and recovery

Caetano Sauer*
Tableau Software
Munich, Germany
csauer@tableau.com

Goetz Graefe
Google Inc.
Madison, WI, USA
goetzg@google.com

Theo Härder
TU Kaiserslautern
Kaiserslautern, Germany
haerder@cs.uni-kl.de

ABSTRACT

Recovery is an intricate aspect of transaction processing architectures. In its traditional implementation, recovery requires the management of two persistent data stores—a write-ahead log and a materialized database—which must be carefully orchestrated to maintain transactional consistency. Furthermore, the design and implementation of recovery algorithms have deep ramifications into almost every component of the internal system architecture, from concurrency control to buffer management and access path implementation. Such complexity not only incurs high costs for development, testing, and training, but also unavoidably affects system performance, introducing overheads and limiting scalability.

This paper proposes a novel approach for transactional storage and recovery called FineLine. It simplifies the implementation of transactional database systems by eliminating the log-database duality and maintaining all persistent data in a single, log-structured data structure. This approach not only provides more efficient recovery with less overhead, but also decouples the management of persistent data from in-memory access paths. As such, it blurs the lines that separate in-memory from disk-based database systems, providing the efficiency of the former with the reliability of the latter.

1. INTRODUCTION

Database systems widely adopt the write-ahead logging approach to support transactional storage and recovery. In this approach, the system maintains two forms of persistent storage: a log, which keeps track and establishes a global order of individual transactional updates; and a database, which is the permanent store for data pages. The former is an append-only structure, whose writes must be synchronized at transaction commit time, while the latter is updated asynchronously by propagating pages from a buffer pool into the database.

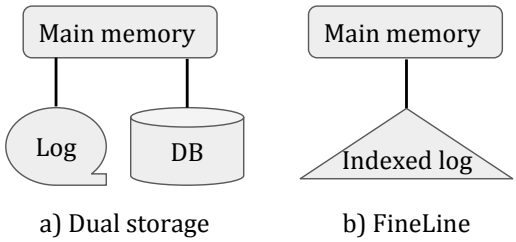

Figure 1: Single-storage principle of FineLine

The primary reason behind such a *dual-storage* design, illustrated in Fig. 1a, is to maximize transaction throughput while still providing transactional guarantees, taking into account the characteristics of typical storage architectures. The log, being an append-only structure, is *write-optimized*, while the database maintains data pages in a "ready-to-use" format and is thus *read-optimized*.

The downside of the dual-storage approach is that maintaining transactional consistency requires a careful orchestration between the log and the database. This requires complex software logic that not only leads to increased code maintenance and testing costs, but also unavoidably limits the performance of memory-resident workloads. Furthermore, the propagation of logged updates into the database usually lags behind the log by a significant margin, leading to longer outages during recovery from system failures.

These problems are known for a long time, as noted recently by Stonebraker, referring to the now 30-year old POSTGRES design [46]:

> *"... a DBMS is really two DBMSs, one managing the database as we know it and a second one managing the log."*
>
> Michael Stonebraker [47]

A *single-storage* approach eliminates these problems, but existing solutions are usually less reliable and more restrictive than traditional write-ahead logging—for instance, because they require all data to fit in main memory, are designed for non-volatile memory and cannot perform well with hard disks and SSDs, or do not support media recovery.

*Work done partially at TU Kaiserslautern

Proceedings of the VLDB Endowment, Vol. 10, No. 9

This paper proposes a novel, single-storage approach for transactional storage and recovery called FineLine. It employs a generalized, log-structured data structure for persistent storage that delivers a sweet spot between the write efficiency of a traditional write-ahead log and the read efficiency of a materialized database. This data structure, referred to here as the *indexed log*, is illustrated in Fig. 1b.

The indexed log of FineLine decouples persistence concerns from the design of in-memory data structures, thus opening opportunity for several optimizations proposed for in-memory database systems. The system architecture is also general enough to support a wide range of storage devices and larger-than-memory datasets via buffer management. Unlike many existing proposals, this generalized approach not only improves performance but also provides all features of write-ahead logging as implemented in the ARIES family of algorithms—e.g., media recovery, partial rollbacks, index and space management, fine-granular recovery with physiological log records, and support for datasets larger than main memory. In fact, the capabilities of FineLine go beyond ARIES with support for on-demand recovery, localized (e.g., single-page) repair, and no-steal propagation, but with a substantially simpler design. Lastly, the FineLine design is completely independent from concurrency control schemes and access path implementation, i.e., it supports any storage and index structures with any concurrency control protocol.

In the remainder of this paper, Section 2 discusses related work. Section 3 introduces the basic system architecture and its components. Sections 4 and 5 then expose the details of the logging and recovery algorithms, respectively. Finally, Section 6 provides some preliminary experiments and Section 7 concludes this paper.

2. RELATED WORK

This section discusses related work, focusing on the limitations that the FineLine design aims to overcome. This new approach is not expected to be superior to all existing approaches in their own target domains, but it is unique in which it combines their advantages in a generalized, robust design.

2.1 Single-storage approaches

System designs that provide a single persistent storage structure fall into two main categories: those that eliminate the log and those that eliminate the database. Before discussing these in detail, it is important to point out that this work considers single and dual storage from a logical data-structure perspective. In other words, a single-storage approach is not one that maintains all data in a single physical device or class of devices, but rather one in which a single data structure is used to manage all persistent data.

Approaches that completely eliminate the log must employ a *no-steal/force* mechanism [25], i.e., they must synchronously and atomically propagate all updates made by a transaction at commit time and uncommitted updates must not reach the materialized database. Early research on transaction processing has proposed some such designs [5, 24, 32], but they never made it into production use because of their inefficiency in traditional storage architectures. The POSTGRES system [46] mentioned earlier falls into the same category, with the main difference that a small amount of non-volatile memory is assumed. Some designs for in-memory databases employ a *no-steal/no-force* strategy [11, 33, 48], which eliminates persistent undo logging, but a log is still required for redo recovery.

The LogBase approach [50] uses a log-based representation for persistent data. The idea is to maintain in-memory indexes for key-value pairs where the leaf entries simply point to the latest version of each record in the log. This way, updates and inserts on the index are random in main memory but sequential on disk. In order to deliver acceptable scan performance, the persistent log is then reorganized by sorting and merging, similar to log-structured merge trees (LSM-trees) [37].

Despite the single-storage advantage, this approach has three major limitations. The first limitation stems from the fact that the approach seems to be targeted at NoSQL key-value stores rather than OLTP database systems. LogBase is designed for write-intensive, disk-resident workloads, and thus it makes poor use of main memory by storing the whole index in it and requiring an additional layer of indirection—and thus additional caching—to fetch actual data records from the log. Furthermore, it destroys clustering and sequentiality properties of access paths, making scans quite inefficient and precluding the use of memory-optimized index structures—e.g., the Masstree [34] used in Silo [48] or the Adaptive Radix Tree [31] used in HyPer [29].

The third limitation is that recovery requires a full scan of the log to rebuild the in-memory indexes. As a remedy, the authors propose taking periodic checkpoints of these indexes. However, these checkpoints essentially become a second persistent storage representation, and the single-storage characteristic is lost. While details of how recovery is implemented are not provided in the original publication [50], it seems like traditional recovery techniques like ARIES [35] or System R [21] are required.

Hyder [6] is a distributed transactional system that maintains a shared log as the only persistent storage structure. The design greatly simplifies recovery and enables simple scale-out without partitioning. It also exploits the implicit multi-versioning of its log-structured storage to perform optimistic concurrency control with its novel *meld* algorithm. Despite the design simplicity and scalability, its recovery and concurrency control protocols are tightly coupled. Furthermore, a fundamental assumption of Hyder is that the entire database is modelled as a binary search tree. These restrictions, while acceptable for the distributed scenario envisioned by the authors, have major performance implications for single-node, memory-optimized database systems. In this case, a single-storage approach that (i) accommodates arbitrary in-memory storage and index structures, and (ii) is independent of concurrency control algorithms would provide more choices for optimizing performance.

Another recent approach for a singe-storage, log-structured database is Aurora [49]. Also aimed at distributed transactions, the goal of this design is to cut down I/O costs of replication and durability, as well as to simplify recovery. It achieves that by shipping log records, grouped by some page-ID partitioning scheme, from processing nodes to replicated storage nodes, where these logs are saved for durability and also applied in local materialized copies of database pages. From a general perspective, a key architectural benefit of the approach is the decoupling of processing and storage via logging, which is also a central goal of this paper. Here, a more general architecture is presented and contrasted with write-

ahead logging for general-purpose database systems, which is also applicable for single-node scenarios and compatible with many existing designs (e.g., by adding a transactional persistence layer to an existing in-memory database).

2.2 Recovery in in-memory databases

Achieving an appropriate balance between reliability in the presence of failures and high performance during normal processing is one of the key challenges when designing transaction processing systems. In traditional disk-based architectures, ARIES [35] and its physiological logging approach seem to be the most appropriate solution, given its success in production. However, the emergence of in-memory databases has since challenged this design.

One prominent alternative to ARIES is to employ logical logging, in which high-level operations are logged on coarser objects, such as a whole table, instead of low-level physical objects such as pages. The advantage of logical logging is that log volume, and therefore traffic to the persistent log device, is significantly reduced, thus increasing transaction throughput. However, a known trade-off of logging and recovery algorithms is that logical logging requires a stronger level of consistency on the persistent database [25], and maintaining such level of consistency may, in some cases, outweigh the benefits of logical logging.

An example of logical logging is found, for instance, in the H-Store database system and its *command logging* approach [33]. Each log record describes a single transaction, encoded as a stored-procedure identifier and the arguments used to invoke it. Because arbitrarily complex transactions are logged with a single log record, the granularity of logging and recovery is the coarsest possible: the whole database. Consequently, the persistent database must be kept at the strongest possible level of consistency—transaction consistency [25]—and maintaining it requires expensive checkpointing procedures that rely on shadow paging [21] or copy-on-write snapshots [33, 29]. The CALC approach [39] is a more sophisticated checkpointing technique that minimizes overhead on transaction processing by establishing virtual points of consistency with a multi-phase process. However, its overhead is still noticeable, as copies of records must be produced and bitmaps must be maintained during a checkpoint; thus, transaction latency spikes are still observed during checkpoints. Because maintaining transaction-consistent checkpoints is expensive, they must be taken sparingly to not affect performance negatively. This, on the other hand, implies that longer recovery times are required in case of failures—not only because the persistent state is expected to be quite out of date, but also because log replay requires re-executing transactions serially [33].

One commonly overlooked aspect of command logging is that it does not really eliminate the overhead of generating physiological log records, since these are still required for transaction abort [33]. Rather, it eliminates the overhead of inserting these log records in a centralized log buffer and flushing them at commit time. These overheads, however, can also be mitigated in physiological logging approaches, e.g., with well-balanced log bandwidth [48], scalable log managers [28], and distributed logging [51]. With these concerns properly addressed, the advantages of command logging seem less compelling.

Given these limitations, a solution based on physiological logging—perhaps more general and susceptible to main-memory optimizations than traditional ARIES—seems more appropriate to provide efficient recovery with minimal overhead on transaction execution.

2.3 Instant recovery

Instant recovery [17] is a family of techniques that builds upon traditional write-ahead logging with physiological log records to improve database system availability. The main idea is to perform recovery actions incrementally and on demand, so that the system can process new transactions shortly after a failure and before recovery is completed.

The instant recovery algorithms—instant restart after a system failure, instant restore after a media failure—exploit the independence of recovery among objects that is inherent to physiological logging [35]. Pages can be recovered independently—and thus incrementally and on demand—during redo recovery by retrieving and replaying the history of log records pertaining to each page independently. The same independence of recovery applies for transactions that must be rolled back for undo recovery. The *adaptive logging* approach [52] exploits this independence to improve recovery times in the command logging technique by adaptively incorporating physiological log records.

The support for incremental and on-demand recovery is crucial for decoupling downtime after a failure from the amount of recovery work that must be performed. In this new paradigm, a more useful metric of recovery efficiency is how quickly the system is able to regain its full performance after a failure, thus unifying concerns of fast recovery and quick system warm-up [38].

2.4 Modern storage hardware

While the cost of DRAM has decreased substantially in the past years, another recent phenomenon in memory technology is the advancement of non-volatile memory devices. Although these efforts seem promising, assuming that the new technology will replace all other forms of storage is likely unrealistic. A relevant example is the advent of flash memory: rather than completely replacing magnetic hard disks, it is employed as a better fit for a certain type of demand, namely low I/O latency and lower (albeit not nonexistent) imbalance between sequential and random access speeds. However, in terms of sustained sequential bandwidth, endurance, and capacity, magnetic disks still provide the more economically favorable option. Therefore, it is reasonable to expect that non-volatile memory will, at least at first, coexist with other forms of storage media, fulfilling certain, but not all, application demands in a more cost-efficient manner.

The "five-minute rule", introduced three decades ago [22] and refined multiple times since then [20, 16], points to the importance of considering economic factors when planning memory and storage infrastructures. A recent re-evaluation of the five-minute rule [2], for instance, makes a strong case for the use of flash-based SSDs in the foreseeable future, potentially coexisting with non-volatile memory. Similar arguments have been made in support for buffer management in modern database systems [19, 30]. The emergence of new memory technologies is likely to further promote the importance of the five-minute rule, leading to a wider spectrum of choices and optimization goals. Thus, systems that embrace the heterogeneity of storage hardware with a unified software approach are desirable.

2.5 Log-structured storage

Much like the recovery log in a write-ahead logging approach with a no-force policy [25], an LSM-tree [37] increases I/O efficiency by converting random writes into sequential. The difference is that the recovery log usually serves a temporary role: during normal operation, log space is recycled as pages are propagated [44]; during restart, at least in a traditional ARIES-based approach [35], the log is mostly read sequentially from pre-determined offsets in the log analysis, redo, and undo phases. Given these restricted access patterns, a typical recovery log is never indexed or reorganized like an LSM-tree.

Our previous work on instant recovery from media failures [17] introduces an indexed, incrementally reorganizable data structure to store log records. Closely resembling an LSM-tree and other forms of incremental indexing, the sequential recovery log is organized into partitions sorted primarily by page identifier and secondarily by LSN. This organization enables efficient retrieval of the history of updates of a single page as well as bulk access to contiguous pages. The instant restore technique [43] makes use of these features in the context of recovery from media failures, but the indexed log data structure has applications far beyond media recovery.

One important caveat of most log-structured approaches is that they rely on a separate write-ahead log [7] to provide ACID semantics. Given that the log itself can be reorganized and indexed, as done in instant recovery [17, 42], there is potential to unify the data structures used to store the log and those used to store data pages on persistent storage. In other words, applying techniques of LSM-trees to store and organize transaction logs seems to be an effective way to eliminate database storage and provide a single-storage transactional system.

Despite both having a log-structured storage component, FineLine differs from LSM-trees in some fundamental ways—perhaps the most notable one being the indexing of physiological log records pertaining to a database page rather than key-value pairs in the application domain. As explained in the remainder of this paper, this approach is key in completely decoupling in-memory data structures from their persistent representation and thus optimizing the performance of in-memory workloads. Section 6.6 empirically evaluates an LSM-tree in comparison with FineLine.

2.6 Summary of related work

In the wide design space between traditional disk-based and modern in-memory database systems, we identify four major design goals that none of the existing approaches seems to fulfill in a holistic and generalized manner. These are: simplified and loosely coupled logging and recovery architecture with a single-storage approach, performance comparable to in-memory databases, efficient recovery following the instant recovery paradigm, and independence from underlying memory technology in favor of heterogeneous and economically favorable infrastructures. We also identify log-structured techniques as a potential approach in designing a single-storage architecture that eliminates traditional database storage. FineLine aims to explore this potential and fulfill the design goals laid out above, hopefully blurring many of the sharp lines that separate different classes of database system architectures.

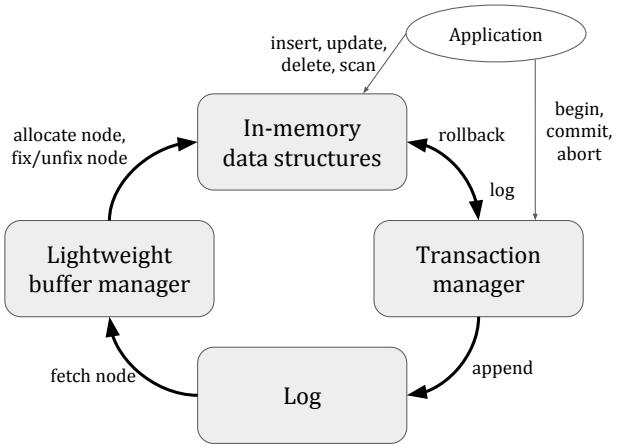

Figure 2: Overview of the FineLine architecture

3. ARCHITECTURE

Fig. 2 shows the four main components of FineLine. This section provides an overview of these components and their interaction.

Applications interact primarily with two components: in-memory data structures and the transaction manager. The latter is used for establishing transaction boundaries with *begin* and *commit* calls, as well as to voluntarily *abort* an active transaction. In-memory data structures are essentially key-value stores, usually storing uninterpreted binary data; a typical example is a B-tree. These are explicitly qualified as *in-memory* because their internal structure is not mapped directly to objects on persistent storage. Instead, their persistent representation is maintained exclusively in the *log* component. Note that there is no database storage, and thus the log serves as single storage structure for all persistent data. Finally, a *lightweight buffer manager* component manages the memory used by data structures and controls caching of data objects.

3.1 In-memory data structures

FineLine is designed to provide persistence to any collection of arbitrary storage structures that can be organized into *nodes* uniquely identified by a *node ID*. It can be used as the storage component of a relational database system—supporting either row- or column-based storage—as well as a generic software library that provides persistence to in-memory data structures with transactional guarantees.

The distinguishing feature of FineLine in contrast to existing approaches is that it provides persistence without mapping data structures directly to a persistent storage representation. This is illustrated in Fig. 3, which compares the propagation of changes from volatile to persistent storage, as well as the retrieval of data in the opposite direction, in traditional write-ahead logging (WAL; top) and in FineLine (bottom). A typical WAL-based database system propagates changes to persistent storage by flushing a node of a data structure into a corresponding page slot on disk. Because this propagation happens asynchronously (i.e., following a no-force policy [25]), these changes must also be recorded in a log for recovery. Following the WAL rule, a log record must be written before the affected page is written. FineLine, on the other hand, never flushes nodes

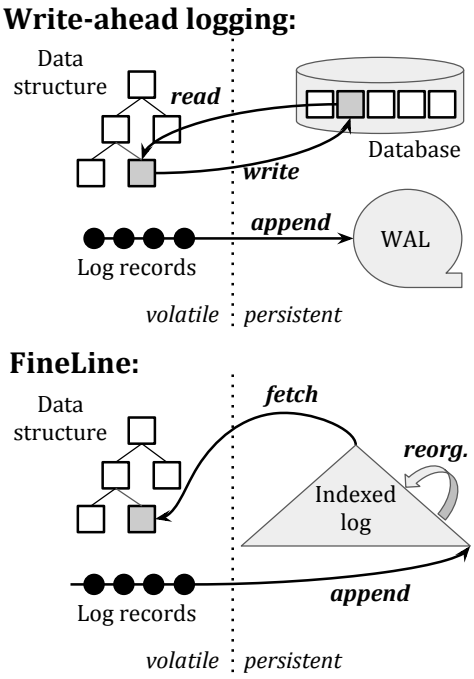

Figure 3: Propagation in WAL (top) vs. FineLine (bottom)

or any other part of an in-memory data structure. Instead, it relies on the log, which is indexed for fast retrieval, as the only form of propagation to persistent storage. In order to retrieve a node into main memory, its most recent state is reconstructed from the log with the *fetch* operation. Sections 3.2 and 4 below discuss the details of the FineLine log, referring back to Fig. 3.

The assumption that data structures are not mapped to persistent storage has profound implications for the rest of the system architecture. First, it naturally implements a *no-steal* policy, which eliminates the need for undo recovery and therefore significantly cuts down the amount of logged information as well as complexity. Furthermore, it eliminates bottlenecks of disk-based database systems in memory-abundant environments, because it allows the use of encoding, compression, addressing, and memory management techniques optimized for in-memory performance. Lastly, by decoupling in-memory data structures from any persistence concern, recovery overheads—such as the interference caused by periodic checkpoints and the maintenance of page LSN values and dirty bits—are also eliminated.

In order to provide fine-granular access to data on persistent storage, i.e., in the log, as well as to support incremental recovery from system and media failures, the FineLine architecture employs a generalized form of physiological logging. Instead of being restricted to *pages*, i.e., fixed-length blocks mapped to specific locations on persistent storage, log records pertain to nodes of a data structure. In this generalized definition, a node is any structural unit of an in-memory data structure, regardless of size, memory contiguity, or internal complexity. It may well be mapped directly to a page—which is even advisable to simplify buffer management—and in these cases the terms can be used interchangeably. However, a node can also be of variable length, point to non-inlined fields, and contain arbitrary internal structure. From the perspective of logging and recovery, a node is any structure that has a unique identifier and is the fundamental unit of caching, recovery, and fault containment.

The choice of what constitutes a node naturally yields a continuum of different logging and recovery algorithms from a single generalized template. If a node is simply a page of a storage structure like a B-tree, FineLine behaves much like ARIES and physiological logging, with the added advantage of simplified and on-demand recovery, better performance, and redo-only logging (i.e., no-steal). On the other hand, if a node represents a whole table, FineLine essentially delivers a logical logging mechanism. A node might also be a single record, which is then a form of physical logging (or value logging, as employed in Silo [48]). The key observation here is that these distinctions are essentially blurred under this new generalized model, where all data is kept in a single indexed log.

3.2 Log

The log component is the centerpiece of the FineLine design, and the main contribution of this work. As Fig. 2 illustrates, the log interface provides two primary functions: *append* and *fetch*. An internal *reorganization* function is also implemented to incrementally and continuously optimize the fetch operation; this is similar to merging and compaction in LSM-trees [37, 45, 8]. Internal aspects of the log are discussed in Section 4; this section focuses on the interface to the other components through the append and fetch operations, as illustrated in the bottom part of Fig. 3.

The append function is called by the transaction manager during transaction commit. Instead of appending a log record into the log for every modification, log records of a transaction are collected in a private log buffer and appended all at once at commit time. This is another key distinction to traditional write-ahead logging, and the process will be discussed in detail in Section 4.

The fetch function is used to reconstruct a node from its history of changes in the log. Following a physiological logging approach, each log record describes changes to a single node, and the indexed log must support efficient retrieval of all log records associated with a given node identifier.

3.3 Lightweight buffer manager

Traditional buffer management in database systems has two key concerns: caching and update propagation. In contrast, FineLine employs a *lightweight* buffer manager whose only concern is caching of nodes. This involves: (1) managing memory and life-cycle of individual nodes, (2) fetching requested nodes from persistent storage (i.e., from the log), (3) evicting less frequently- or less recently-used nodes, and (4) providing a transparent addressing scheme for pointers among nodes. The concerns of update propagation, on the other hand, include keeping track of version numbers (i.e., PageLSN values) and durable state (i.e., dirty bits) of individual nodes, propagating changes to persistent storage by writing pages, and enforcing the WAL rule. FineLine eliminates these concerns by performing propagation exclusively via logging.

The FineLine design does not require a specific buffer management scheme; traditional page-based implementations [23]—preferably optimized for main memory [19, 30]—as well as record-based caching schemes [10, 13] are equally

conceivable. It is nevertheless advisable to use nodes as the unit of caching and eviction, since it is a better fit for the rest of the system architecture.

3.4 Transaction manager

The transaction manager component of FineLine is responsible for keeping track of active transactions, implementing a commit protocol, supporting rollback, guaranteeing transaction isolation, and maintaining transaction-private redo/undo logs. The latter aspect is a key distinction of the present design to traditional ones. Operations on in-memory data structures generate log records that are kept in a thread-local, per-transaction log buffer. During commit, all log records generated by a transaction are appended into the log atomically. Section 4 provides details of the commit protocol.

Note that because nodes are never flushed to persistent storage, FineLine implements a *no-steal* policy; this means that only redo information is required in the log. As implemented in most in-memory databases [29, 33], a transaction-private undo log is used for rollback and it may be discarded after commit. In essence, an aborted transaction leaves absolutely no trace of its existence in the persistent state of the database.

Concurrency control among isolated transactions is also an important role of the transaction manager. A two-phase locking approach, for instance, requires a lock manager component (omitted in Fig. 2). The private logs can also be reused for validation in optimistic concurrency control schemes as well as for multi-versioning (e.g., Silo [48] and HyPer [36]). The present design does not assume any specific concurrency control scheme; thus, the transaction manager is treated as an abstract component.

4. LOGGING

The FineLine indexed log serves as the only form of persistent storage, and thus it must support both writes and reads efficiently. Writes happen at commit time with the append function, while reads are performed with the fetch function. To provide acceptable fetch performance, the log is incrementally and continuously reorganized by merging and compacting partitions. This section discusses these operations in detail.

4.1 Indexed log organization

The indexed log is actually a partitioned index, in which probes require inspecting multiple partitions before returning a result [14]. A partitioned index is ideal for the FineLine log because it performs all writes sequentially, like a traditional write-ahead log, at a moderate and, thanks to caching, amortized cost on read performance. As discussed in Section 1, this is precisely the goal of a single-storage approach: combine the write efficiency of the log with the read efficiency of a database file.

Fig. 4 illustrates the partitioned-index log organization used in FineLine in contrast with a traditional sequential log. In this minimal example, three transactions generate log records on two nodes, A and B. In the traditional organization, each operation is assigned an LSN value (omitted here), and their total order must be reflected in the sequential log. In the indexed organization, log records are sorted by node ID within each partition. Log records pertaining to the same node, in turn, must be ordered by a sequence number that reflects the order in which the operations were applied to the node. In the example diagram, each transaction forms a new partition—which would be a naive implementation but illustrates the point well. The order of transactions, and thus the order of partitions, is irrelevant, because a total order is only required among log records of the same node. The reason for that is the absence of undo recovery, as discussed in Section 5. For a comprehensive discussion on the issue of log record ordering, we refer to the work of Wang and Johnson [51] and their GSN approach.

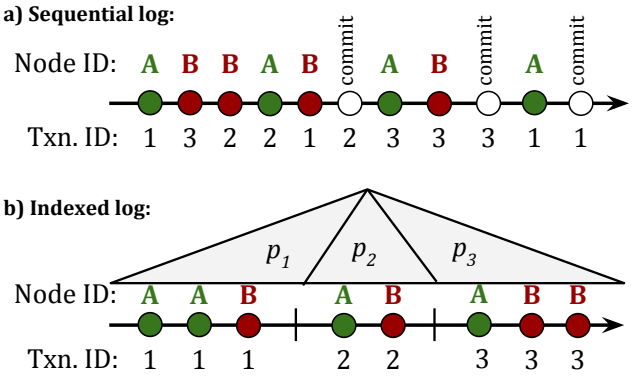

Figure 4: Sequential log (a) and its equivalent indexing (b)

Each append operation, which is invoked by the commit protocol discussed below, creates a new partition in the log. Therefore, the indexed log maintains the append-only efficiency of a sequential log for write operations (i.e., transaction commits). For read operations, i.e., node fetches, the principal design challenge is to effectively approximate the read efficiency of a page-based database file by merging partitions. This is essentially the same challenge faced by all log-structured designs [37, 41, 45], and thus many techniques can be borrowed.

In the example, if node A were to be fetched, each of the three partitions would be probed and merged in order to replay the four log records in the correct order. This example assumes that the first log record represents the creation of the node, so that it is fully reconstructed from log records only. Section 4.3 provides further details of the fetch operation and the reorganization of log partitions. Before that, Section 4.2 discusses how new partitions are created with transaction commits.

4.2 Commit

Commit processing in ARIES (shown in Fig. 5a) maintains a centralized log buffer where a log record is appended for each update of any transaction. These appends must be synchronized to produce a global LSN order, and thus the contention is quite high. In the example, the commit of T_2 causes all contents of the log buffer up to its commit log record to be flushed to persistent storage.

In the FineLine commit protocol, updates to in-memory data structures generate physiological log records that are maintained in transaction-private log buffers. At commit time, a transaction's log buffer is inserted into a system-wide *commit queue*. At this point, here referred to as the *pre-commit*, locks can be released [18] and resources can be freed—the transaction may not rollback anymore and the

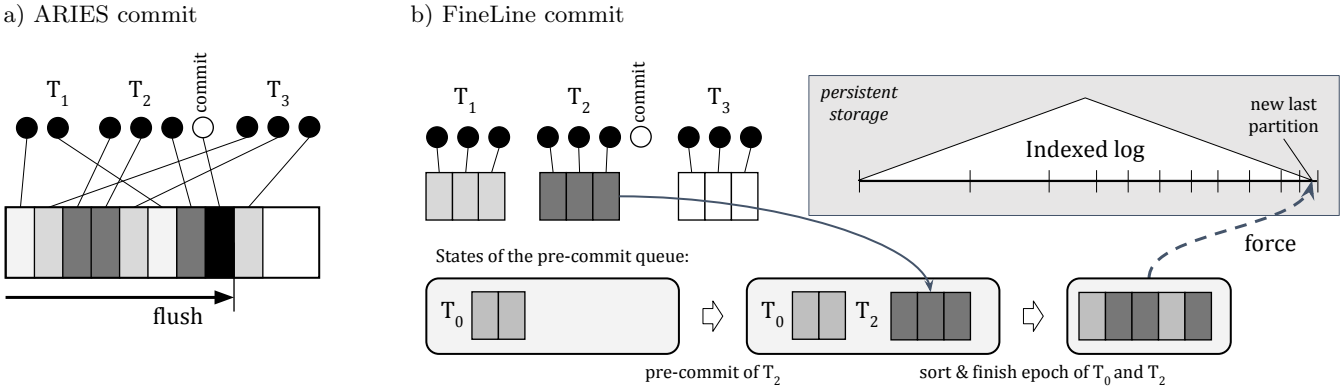

Figure 5: Commit protocols of ARIES (a) and FineLine (b)

commit will be acknowledged as soon as the log records are forced to persistent storage.

The commit queue is formatted as a log page that can be appended directly to the indexed log. Before the append occurs, the log records in this page are sorted primarily by node ID and secondarily by a node-local sequence number.

This sort can be made very efficient if log pages are formatted as an array of keys (or key prefixes) and pointers to a payload region within the page.

Each appended log page creates a new partition in the indexed log[1]. An alternative implementation maintains the last partition unsorted, i.e., it retains the order of transactions established by the pre-commit phase. Since it is unsorted, the last partition can be much larger than a log page. To enable access to the history of individual nodes in this last partition, it must either be sorted or use an alternative indexing mechanism [2]. For ease of exposition, we assume here that each group commit creates a new sorted partition that is directly appended to the log index.

Instead of forcing their own logs, transactions block waiting for a notification from an asynchronous commit service, which is responsible for the final commit, i.e., the "hardening", of transactions. This technique, borrowed from the Aether log manager [28] and also used in the Silo in-memory database [48], essentially implements the group commit approach [11], allowing for effective amortization of I/O costs and reduced system call overhead. As for minimizing the contention to a single log buffer, the consolidation approach used in Aether can also be employed here; note, however, that the contention is significantly reduced, because only one log buffer insert is required per transaction, instead of one per log record.

Each commit group defines an *epoch*, which is basically the unit of transaction hardening and can also be used for efficient resource management [48]. Once an epoch is finished, clients are notified and their commit is finally acknowledged. As with traditional group commit, epochs may be defined by a variety of policies—e.g., fixed time intervals, log volume thresholds, or a combination of both.

The example of Fig. 5b illustrates commit processing in FineLine. It starts with active transactions $T_1 \ldots T_3$, whose updates, represented by black dots, generate log records in a private buffer, represented by rectangles. A transaction T_0 has already pre-committed and thus its log records are already in the pre-commit queue. Then, T_2 enters the pre-commit phase—an event represented here by a white dot—and appends its logs into the pre-commit queue. The commit service then kicks in and starts a new epoch by atomically swapping the pre-commit queue with an empty one. Then, it sorts all logs of the previous epoch into a new partition of the indexed log. After that, the previous epoch is considered durable and the commit of T_0 and T_2 is acknowledged.

For now, we assume a single, system-wide log buffer, but our technique can be extended to multiple commit service threads to provide distributed logging. Such extension is out of the scope of this paper, but existing techniques can be adapted [48, 51].

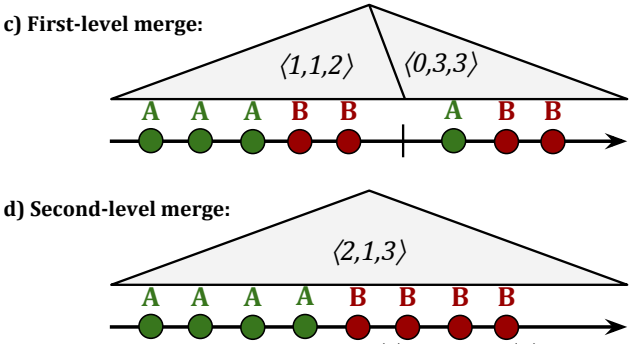

Figure 6: Partitioned log after one (c) and two (d) merges

4.3 Node fetch and merging

In the architecture diagram of Fig. 2, traversing a node pointer in an in-memory data structure may incur a cache miss in the lightweight buffer manager, which requires invoking the fetch operation on the log. To perform a node fetch, each log partition is probed in reverse order, starting from the most recent one, and the log records are merged and collected in a thread-local buffer. This process stops when a *node-format* log record is found, which can be either an initial node construction and allocation or a "backup" image of a node, containing all data required to fully reconstruct it. Then, starting from the node-format log record, all committed updates are replayed in a previously allocated empty node, bringing it to its most recent, transaction-consistent state.

In order to reduce the number of probes and deliver acceptable fetch performance in the long run, partitions are

[1]The choice of log page size depends on a good balance between maximizing write bandwidth, minimizing fragmentation, and optimizing group commit schedules [26].

[2]The experiments in Section 6 use this approach to implement FineLine as an extension of an existing WAL system.

merged and compacted periodically with an asynchronous daemon, like in LSM-trees [37, 45]. To simplify the management of partitions with concurrent node fetches and merges, as well as to enable garbage collection, partitions are identified by a three-component key of the form $\langle level\_number, first\_epoch, last\_epoch \rangle$. Initial partitions generated by the commit protocol have level zero, and they are numbered sequentially according to their epoch number e, so their identifiers are of the form $\langle 0, e, e \rangle$. A merge of partitions of level i yields a partition $\langle i+1, a, b \rangle$, where a is the lowest $first\_epoch$ of the merged partitions and b is the highest $last\_epoch$. Only consecutive partitions may be merged, so that all epochs in the interval $[a, b]$ are guaranteed to be contained in the resulting partition.

Fig. 6 illustrates the merge process, where the two states (c and d) of the index are derived from the states (a and b) of Fig. 4. In the first merge, partitions p_1 and p_2, whose proper notation is $\langle 0, 1, 1 \rangle$ and $\langle 0, 2, 2 \rangle$, are merged into a level-one partition $\langle 1, 1, 2 \rangle$. The second merge then produces a single partition $\langle 2, 1, 3 \rangle$ using p_3 (i.e., $\langle 0, 3, 3 \rangle$).

When performing a fetch operation, a list of partitions to be probed can be derived by computing maximum disjoint $[first\_epoch, last\_epoch]$ intervals across all existing levels (i.e., the smallest set of partitions that covers all epochs). This can occur either by probing the index iteratively or relying on auxiliary metadata—such implementation details are omitted here. Furthermore, while not illustrated in the examples, the creation of a new partition through merging does not overwrite the merged lower-level partitions—these can be garbage collected later on. We refer to previous work on partitioned B-trees for a thorough coverage of these techniques [14].

While the process of probing the partitioned index for log records and reconstructing a node is fairly simple, it may be inefficient if implemented naively, especially because the commit protocol is expected to produce multiple partitions (up to hundreds or thousands) per second. Therefore, the following paragraphs discuss techniques to optimize node fetches and mitigate such performance concerns.

As a first and foremost optimization, pages of the indexed log must be cached in main memory for efficient retrieval. This ensures that index probes incur either reads on leaf pages only or no reads at all for the most recent partitions. Second, system-maintained constraints and auxiliary data structures such as Bloom filters, succinct trees [53], or zone indexes [15] can be used to avoid probing partitions that are known to not contain a given node identifier. These two measures can substantially reduce the number of read operations required for a single node fetch.

When merging partitions, opportunities for compaction should be exploited. In the FineLine context, compaction can be characterized as follows: given a stream of log records $l_1 ... l_n$ pertaining to the same node, produce a stream of log records $k_1 ... k_m$ such that the total size in bytes is reduced. Other than simple compression techniques like omitting the node identifier or applying delta encoding to keys inside each log record, a more effective compaction strategy involves node-format log records. If any of l_i $(1 \leq i \leq n)$ is a node-format log record, then a single node-format log record k can be produced (i.e., $m = 1$) for the whole stream. This log record essentially creates a snapshot of the node as reconstructed from l_i and all subsequent updates until l_n. Such node-format log records deliver the best-possible read efficiency for the fetch operation, with the same cost as a single database page read in a traditional WAL approach.

Note that by employing effective node eviction policies for in-memory data structures, high-cost fetch operations requiring tens or hundreds of partition probes should be extremely rare. This is because the buffer manager should absorb the vast majority of reads when sufficient main memory is available. Therefore, the vast majority of node fetches should be of cold nodes. Since cold nodes are expected to not have been updated recently, it is very likely that all relevant log records have been merged into higher-level partitions, possibly into a single node-format log record. Such higher-level partitions, containing one node-format log record for most nodes, can be viewed as a generalized form of database storage as employed in a dual-storage architecture.

5. RECOVERY

The design and implementation of recovery algorithms must take into consideration the different levels of abstraction in a system architecture and the degree of consistency expected between them [25]. Traditional ARIES recovery, for instance, restores database pages to their most-recent state and rolls back persisted updates made by loser transactions. This process involves scanning the log in three phases—analysis, redo, and undo. To work properly, the algorithm must "trust" the contents of the log file, and thus it relies on a certain degree of consistency provided by the file system. Within its layer of abstraction, the file system may employ its own recovery measures to guarantee such consistency, and these are hidden from the database system.

In contrast with ARIES, higher-level recovery in FineLine is significantly simpler—in fact, as discussed later on, there is actually no code path which is specific to recovery in a traditional sense. However, this simpler, almost transparent mechanism requires a degree of consistency from the indexed log data structure that cannot be delivered by file systems alone. To better explain how recovery works in FineLine, the following sub-sections explicitly separate these concerns into two levels of consistency: that expected from the internal indexed log data structure, and that expected from the transaction system as a whole (i.e., full ACID compliance). Lastly, a final sub-section discusses concerns of stable storage and media failures.

5.1 Indexed log consistency

As discussed in Section 3, the log interface provides two operations to other components: *append*, which appends one or more log pages to a new partition in the index, and *fetch*, which is essentially a sequence of index probes on each partition. Furthermore, incremental maintenance of the index requires adding a new partition when performing a merge and deleting a set of partitions during garbage collection (e.g., after a merge). The fundamental requirement for consistency is that these operations must appear atomic.

In order to support the atomicity of index operations, multiple design and implementation choices are conceivable, and discussing them would be beyond the scope of this paper. Nevertheless, it may be worthwhile to point out that the indexed log has a particular access pattern which is more restricted than that of a relational index in main memory, for instance. Most importantly, there are no random updates—the commit protocol appends whole log pages and merging

creates whole partitions at once. This makes designs based on shadow paging much more attractive; for instance, a copy-on-write B-tree [3, 40] would be perfectly suitable. Another approach would be to store each partition as a sequential file on disk and maintain all index information in main memory; a caveat, however, is that restoring this index information after a system failure substantially adds to the total recovery time, whereas a copy-on-write B-tree has practically instantaneous recovery [3].

5.2 Transaction consistency

Provided that the indexed log is kept consistent with atomic operations, no additional steps are required to perform recovery after a system failure, i.e., transactions can start immediately after the system boots up. This is because the node fetch protocol already replays updates up to the most recent committed log record. Furthermore, no undo actions are required because of the no-steal policy.

To understand why explicit recovery is not required, consider the steps involved in the three phases of ARIES recovery and why they are required. The goal of the first phase—log analysis—is to basically determine what needs to be redone (i.e., dirty pages and their dirty LSN) and what needs to be undone (i.e., loser transactions to be rolled back) [35]. In a no-steal propagation scheme, there is no undo phase and therefore log analysis and checkpoints would only be concerned with dirty pages that might require redo. In preparation for the redo phase, ARIES recovery requires determining the dirty LSN—i.e., the LSN of the first update since the page was last flushed—of each dirty page. This is not required in FineLine because a node fetch automatically replays all committed updates without any additional information. Another way to look at this is that whatever information would be collected during log analysis (e.g., a dirty-page table) is embedded—and maintained atomically—in the indexed log itself.

As for availability in the presence of system failures, this recovery scheme naturally supports incremental and on-demand recovery, much like the instant restart algorithm [17]. However, FineLine goes beyond instant restart by eliminating an offline log analysis phase. Thus, the actual downtime after a failure depends solely on the time it takes to boot-up the system again. A much more useful metric of recovery efficiency, in this case, is how quickly the system regains its "full speed", i.e., the pre-failure transaction throughput. Here, techniques such as fetching nodes in bulk and proactively merging partitions with high priority have a substantial impact. Also note that the concerns of efficient recovery are essentially unified with concerns of buffer pool warm-up [38]. Unfortunately, discussing such techniques is out of the scope of this introduction to FineLine.

Given that recovery happens as a side-effect of fetching nodes, and that no offline analysis is required, this also implies that checkpoints are not required during normal processing. This reduces not only overheads and interference on running transactions, but also—and perhaps most importantly—code complexity and architectural dependencies. Instead, the asynchronous merge and compaction of partitions is what shortens recovery time, speeds up warm-up after a restart, and improves node fetch performance. In fact, these three concerns are essentially the same in the generalized approach of FineLine.

Whether this novel recovery scheme actually has no recovery actions or the recovery actions are embedded in normal operations is simply a matter of perspective. As emphasized before, the single-storage approach of FineLine is in fact a generalization of sequential logs and database pages. In this architecture, there is no meaningful distinction between normal and recovery processing modes.

5.3 Stable storage and media failures

Recovery algorithms based on write-ahead logging rely on the *stable storage* assumption, i.e., that contents of the log are never lost. This is, of course, an ideal concept used simply for abstraction in theory. In practice, it basically means that the log must be kept on highly reliable—and expensive—storage. Given that the "head" of the sequential log is expected to be recycled quickly as transactions are finished and updates are propagated to their final destination in the database, the log device can be fairly small, so the high reliability cost is usually not a concern.

While the FineLine logging and propagation mechanisms are quite different from typical write-ahead logging designs, the concept of stable storage is still required, and it would be implemented in practice using exactly the same measures. In FineLine, log partitions of level zero, i.e., partitions generated by commit epochs and not yet merged, must be kept on stable storage. These partitions would then be eligible for garbage collection as soon as they have been merged into level-one partitions. Note, therefore, that the equivalent of propagating updates to the database is, in the FineLine approach, merging partitions.

This storage management approach using log partitions also replaces—or rather generalizes—backup and recovery strategies for media failures. While details are out of the scope of this paper, we briefly discuss the techniques involved. In ARIES, media recovery essentially requires archive copies of both the log and the database [35]. If the database device fails, a backup image is first restored in a replacement device (which may involve loading full and incremental backups), and then the log archive is replayed on the restored pages. Using an indexed log archive, which is actually quite similar to the FineLine log, instant restore [43] performs these operations incrementally and on demand, incurring mostly sequential I/O. In FineLine, a similar restore procedure can be employed, but instead of restoring a database device by merging a backup and a log archive, lost partitions can be recovered by re-merging lower-level partitions or replicas of the lost partitions.

We emphasize that, in comparison with traditional write-ahead-logging recovery, FineLine is by no means less reliable. While the concepts and algorithms are quite different in this generalized approach, the level of reliability depends solely on the hardware infrastructure and the level of redundancy of persistent storage; whether this redundancy takes the form of log archive plus backup images or partitions that are copied and merged does not affect the level of reliability. In fact, the FineLine approach enables much more cost-effective solutions than ARIES while achieving the same reliability. As argued in previous work on instant restore [43], the substantial reduction in mean time to repair presents two attractive options: to either increase availability without additional operational cost; or to maintain the same level of availability with reduced operational cost.

6. EXPERIMENTS

This section describes some experiments performed to investigate the feasibility of the FineLine approach. We start with a description of the prototype implementation and a discussion of the hypotheses that we aim to prove or refute. Then, we present four experiments that compare the log-structured, single-storage approach with a traditional WAL system as well as an LSM-tree.

6.1 Implementation

To evaluate the feasibility of the single-storage approach of FineLine, we have adapted the Zero storage manager[3]—a fork of Shore-MT [27] that implements instant restart and instant restore [17, 43]—to eliminate the materialized database and perform logging as described in Section 4. The prototype aims to approximate the FineLine design with an existing WAL system, performing as little implementation changes as possible. This also implies that the performance comparisons with the baseline system provide a fair evaluation of traditional WAL vs. FineLine, sharing a large percentage of common code and infrastructure.

The commit protocol described in Section 4.2 is implemented in a slightly different manner, which aims to reuse the same log manager implementation as the baseline system [28]. Rather than immediately appending each epoch to the indexed log as a new partition, a suffix of the log is maintained unsorted—or rather, sorted by order of commit. Sorting and indexing run constantly in the background, so that this unordered suffix is kept fairly small. Because index fetches are only supported from the sorted partitions of the log, eviction of a node from the buffer pool must wait until all log records affecting that node have been sorted and indexed; this is easily achieved by maintaining an epoch field on each node, analogous to the PageLSN of ARIES [35]. To provide earlier availability during recovery, node fetches may also replay log records in the unsorted partition using sequential scans—this presents a trade-off between time to first transaction and overall recovery speed.

Unlike logging in ARIES, the FineLine prototype described above only logs redo information and appends all log records of a particular transaction in a single step during commit, as described in Section 4.2. Furthermore, because there is no persistent database, there are no checkpoints of any kind; instead, partitions are merged in the background to gradually improve both the performance of node fetches and the effort of recovery in case of a failure, as well as to recycle log space.

6.2 Hypotheses

This paper focuses on presenting FineLine as a viable architectural alternative to traditional WAL systems such as ARIES. Therefore, our main goal in these experiments is to demonstrate that the single-storage approach of FineLine is capable of delivering better performance than a WAL system, thanks to reduced logging overhead and a more efficient commit protocol, but with a simpler architecture and retaining the advantages of transparent support for larger-than-memory workloads and efficient recovery. Furthermore, we show that recovery performance is drastically improved by providing on-demand, incremental recovery from system failures, as in the instant recovery approach [17, 42]. Lastly, we evaluate transaction performance as the workload size grows beyond the size of main memory and evaluate the trade-offs in comparison to LSM-trees.

More specifically, our experiments test the following hypotheses, each with a dedicated sub-section below:

1. The FineLine prototype improves performance—as measured by transaction throughput—in comparison with a WAL system, thanks to the lack of persistent undo logging and the new commit protocol.

2. The FineLine prototype is able to recover from a system failure and warm up the buffer pool with comparable performance to a state-of-the-art WAL system with support for instant recovery. In comparison with traditional WAL, it should recover much faster.

3. For larger-than-memory workloads, the performance of the FineLine prototype is still better than in a WAL system, but as the data set grows—or, equivalently, as the buffer pool size decreases—a turning point is expected where WAL performs better.

4. The application scenarios that favor LSM-trees also favor FineLine and the same trade-offs can be achieved by tuning system parameters or design choices.

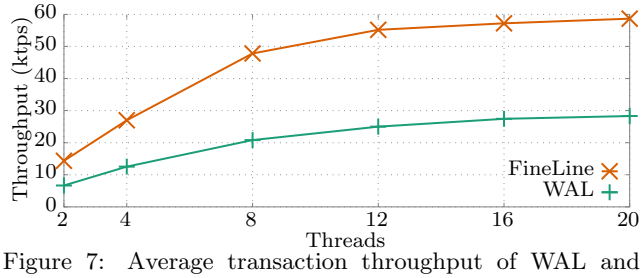

Figure 7: Average transaction throughput of WAL and FineLine with increasing worker thread count

6.3 Transaction throughput

The workload used here consists of the TPC-C benchmark with scale factor 75, which produces ∼10 GB of initial database size. All TPC-C experiments shown in this paper were executed with up to 20 worker threads and Samsung 840 Pro SSDs as persistent storage devices.

The experiment of Fig. 7 shows average transaction throughput (y-axis) for a 5-minute execution of the TPC-C benchmark with increasing worker-thread count (x-axis) on both the WAL system and FineLine. As the results show, throughput is roughly increased by a factor 2 across all worker-thread counts. In order to test in-memory performance only, all these executions employ a buffer pool larger than the total dataset size. Furthermore, because both systems share the same implementation of benchmark components other than logging and recovery, the scalability curve looks very similar.

To confirm that the observed performance improvement is a consequence of the reduced log volume without undo logging and the more efficient commit protocol, Fig. 8 shows the average log size (left side) as well as the average number of log buffer insertions (right side) per transaction for the experiment above. The log volume per transaction is reduced in FineLine by roughly 45%, which is attributed to the elimination of undo information. Note that this

[3] http://github.com/caetanosauer/fineline-zero

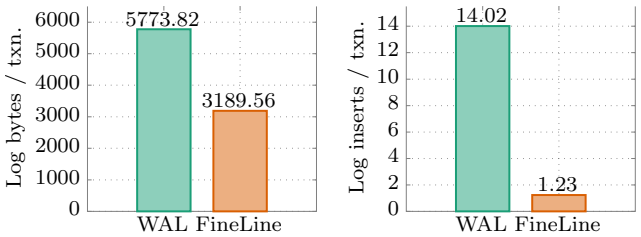

Figure 8: Average log bytes and insertions per transaction in WAL and FineLine for the TPC-C benchmark

includes not only before-images of data values but also log-record metadata such as transaction ID, undo-next pointer, relation ID (for logical undo), etc. The number of log inserts per transaction is drastically reduced in FineLine to less than 9% of that of the WAL system; this is because one log buffer insertion is performed per transaction rather than per log record. The value computed here is higher than one because the experiment includes log insertions of both system transactions as well as the user transactions that initiate them. For instance, if a user transaction triggers the split of a B-tree node, the split has to be logged independently of the fate of the user transaction; thus, it is inserted in the log buffer before the pre-commit of the user transaction. Other system actions, such as page allocations, space-management tasks, and logged events, are also accounted for in this manner, which explains why FineLine generates 1.23 insertions per user transaction.

The experiments above confirm our first hypothesis postulated earlier, namely that redo-only logging and the efficient commit protocol greatly improve the performance of the logging subsystem, delivering higher transaction throughput.

6.4 Restart and warm-up

The restart experiment measures the time it takes to recover and warm-up the buffer pool, i.e., to reach maximum, steady-state transaction throughput, after a system failure. To set up this experiment, the TPC-C benchmark is loaded and ten million transactions are executed, after which the system abruptly shuts down. During this phase, standard propagation measures are enabled in each system, namely buffer-pool write-back in WAL and background partition merging in FineLine. Then, the benchmark restarts and the average transaction throughput (y-axis) is plotted over time (x-axis); this shows how quickly the systems are able to recover and warm-up the buffer pool.

The results are shown in Fig. 9, which compares restart time between WAL, FineLine, and WAL enhanced with instant restart [17] (shown as WAL-Instant). First, by comparing FineLine and WAL, we observe that transactions start executing (and thus the system becomes available) much earlier in the former—namely a few secons vs. more than two minutes. This can be confirmed in Fig. 10, which essentially zooms into the first 40 seconds of the experiment to show when the first transactions are executed. Furthermore, because peak performance is higher in FineLine, warm-up happens at a similar pace but goes twice as higher than in the WAL variant. The re-establishment of peak performance happens roughly one minute earlier in FineLine.

A second comparison can be made in Fig. 9 between FineLine and the WAL system with instant recovery. Because the latter is only offline during the log analysis phase, while redo and undo actions are preformed on demand [17], it

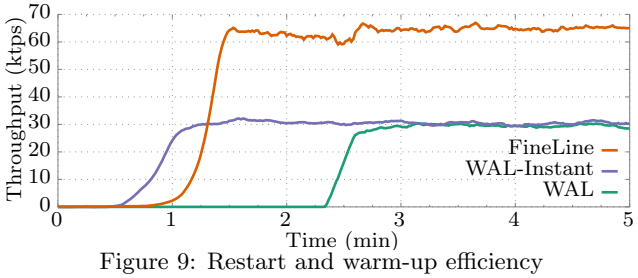

Figure 9: Restart and warm-up efficiency

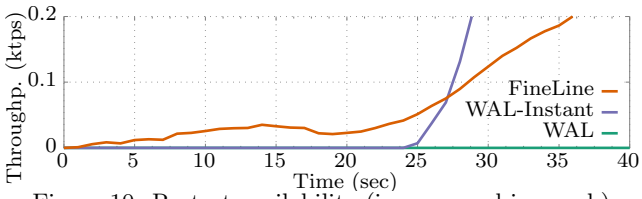

Figure 10: Restart availability (i.e., zoomed-in graph)

is actually able to warm-up faster than FineLine, reaching peak throughput roughly a half minute earlier. FineLine actually has a similar behavior, but warm-up is slightly slower because of the cold index probes required to fetch nodes. This is likely a limitation of the implementation used in our prototype, which, as explained earlier, reuses the same basic logging infrastructure of the WAL system by maintaining the last partition of the log unsorted and then relies on sequential scans to replay updates in the unsorted part.

These results confirm our second hypothesis, namely that recovery performance in FineLine is comparable to a state-of-the-art WAL implementation with instant restart, but much better than a traditional WAL implementation.

6.5 Larger-than-memory workload

The next experiment evaluates the performance of larger-than-memory workloads in FineLine. We analyze average transaction throughput during a three-minute run of the TPC-C benchmark with warmed-up buffer pool. The buffer pool size varies from 10% (1 GB) to 100% (10 GB) of the initial database size. The results are shown in the bar charts of Fig. 11. On the left side, the transaction throughput (y-axis) is plotted for each buffer size (x-axis) of the WAL system; on the right side, the same is shown for FineLine.

The results show that FineLine is quite effective in dealing with limited buffer pool sizes, except for the smaller sizes of 1 and 2 GB, where WAL delivers higher performance. This is expected because as the buffer size decreases, transaction latency is dominated by reads from the database in WAL and, equivalently, indexed log fetches in FineLine. However, thanks to the higher performance of FineLine, higher transaction throughput is observed in the buffer sizes beyond 3 GB. Furthermore, these results indicate that FineLine makes better use of medium-sized buffer pools in relative terms—note, for example, how the 5-GB buffer pool delivers more than 50% the performance of the 10-GB buffer pool, while the same size in the WAL variant delivers less than a third of maximum performance (exact numbers are 7.7 and 24.8 ktps, respectively).

These results confirm our third hypothesis, namely that FineLine still outperforms WAL as the dataset grows beyond main memory, up to a turning point where it becomes slower. Therefore, for large workloads with very scarce main memory, traditional WAL might still be preferred.

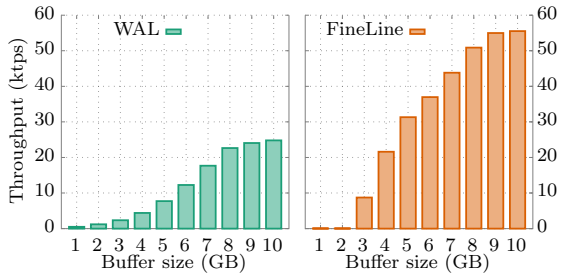

Figure 11: Txn. throughput with varying buffer pool sizes

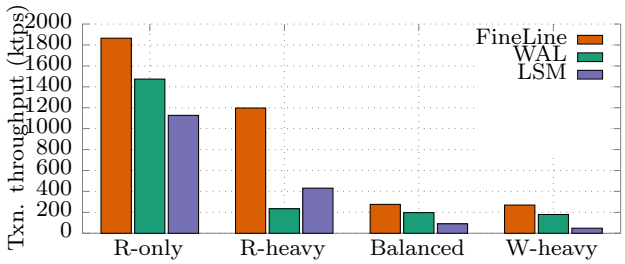

Figure 12: Transaction throughput of the YCSB benchmark

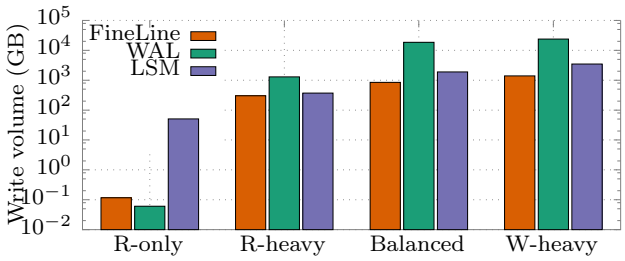

Figure 13: Data volume written in the YCSB benchmark

6.6 Performance comparison with LSM-trees

A thorough comparison of FineLine with LSM-trees and traditional WAL systems would be enough for a separate experiment and analysis paper, given the complexity of design choices [8, 3], hardware configurations [9, 2], and trade-offs [4] involved. Thus, this section focuses on a simple benchmark based on the YCSB dataset to demonstrate pragmatically that the trade-offs offered by LSM-trees in comparison with WAL are also present in FineLine.

To test these hypothesis, we implemented an LSM storage module based on LevelDB [1] in the Shore-Kits framework that runs on top of both Shore-MT/Zero (the WAL system) and FineLine. The reason behind choosing LevelDB is its widespread adoption as an open-source project and thus ease of use and understanding, which aligns with the pragmatic nature of our experiments. A comparison with more advanced and highly tunable LSM engines such as RocksDB [12] is left for future work.

To vary access patterns, we consider a similar setup from the paper of Arulraj et al. [3], which also evaluates log-structured and WAL approaches. The experiment executes 500 million operations on records of 1000 bytes with 8-byte keys; these are either read or updated randomly under four different workloads: read-only (100% reads), read-heavy (90% reads), balanced (50% reads), and write-heavy (10% reads). The storage device used here is a DELL PM1725a NVMe SSD and 20 worker threads are used. The issue of caching and read amplification is not considered here, as the differences between each system are less prominent in comparison with write efficiency; thus, enough main memory is provided to all systems to maximize the cache hit ratio.

Fig. 12 shows the transaction throughput observed for each of the four workloads under the three different engines. Thanks to the lack of logging overheads (including log record generation, insertion into the centralized buffer, and I/O), the read-only workload exhibits by far the best performance in all systems. The first observation is that, in all workloads considered, FineLine provides the highest transaction throughput. An addition of 10% updates, as is the case in the read-heavy workload, causes a significant drop in the WAL system—this result shows how large the impact of logging overheads is, and how effective its optimization can be, as demonstrated by the FineLine result. Here, the LSM-tree is faster the WAL system, but the latter performs better in the balanced and write-heavy workloads, which is a surprising result. This is likely due to a more scalable implementation of the WAL system.

Fig. 13 shows the total data volume written to persistent devices during the experiment. Note that the y-axis is in a log scale here. Except for the read-only scenario, in which LevelDB still performs background compaction, the WAL system exhibits the highest write amplification. This is because it has to write back pages of 64 KB when only a few 1 KB records were modified—note that this was expected, as the experiment is designed to benefit LSM-trees. The write volume increases substantially on all systems as the ratio of updates increases, but their proportions remain similar. Note that FineLine writes less data than the LSM-tree, even though it delivers significantly higher transaction throughput.

7. CONCLUSION

This paper presented a log-structured design for transactional storage systems called FineLine. The design follows a single-storage approach, in which all persistent data is maintained solely in an indexed log data structure, unifying database and recovery log in a more general approach. This novel storage architecture decouples in-memory data structures from their persistent representation, eliminating many of the overheads associated with traditional disk-based database systems.

The log-structured approach also greatly improves recovery capabilities in comparison with state-of-the-art designs. Because the indexed log contains all information required to always retrieve data items in their most recent, transaction-consistent state, recovery from a system failure is fast and makes the system available much earlier. Furthermore, unlike most state-of-the-art approaches, there are no checkpoints, no offline log scans, and no auxiliary data structures that must be rebuilt during recovery.

By mixing existing data management techniques—such as log-structured access methods, in-memory databases, physiological logging, and buffer management—in a novel way, FineLine achieves a design and architecture sweet-spot between modern in-memory database systems and traditional disk-based approaches. It also introduces a new, generalized perspective to data storage and recovery, in which the distinctions between persisted data objects and logs, much like the distinctions between in-memory and disk-based data management, are essentially blurred.

8. REFERENCES

[1] LevelDB. https://github.com/google/leveldb, 2018. Online; accessed 2018-07-12.

[2] R. Appuswamy, R. Borovica-Gajic, G. Graefe, and A. Ailamaki. The Five-minute Rule Thirty Years Later and its Impact on the Storage Hierarchy. In *Proc. ADMS Workshop*, 2017.

[3] J. Arulraj, A. Pavlo, and S. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proc. SIGMOD*, pages 707–722, 2015.

[4] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing access methods: The RUM conjecture. In *Proc. EDBT*, pages 461–466, 2016.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - A transactional record manager for shared flash. In *Proc. CIDR*, pages 9–20, 2011.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[8] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *Proc. SIGMOD*, pages 79–94, 2017.

[9] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. B. Zdonik, and S. Dulloor. A prolegomenon on OLTP database systems for non-volatile memory. In *Proc. ADMS Workshop*, pages 57–63, 2014.

[10] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.

[11] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proc. SIGMOD*, pages 1–8, 1984.

[12] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing space amplification in rocksdb. In *Proc. CIDR*, 2017.

[13] A. Eldawy, J. J. Levandoski, and P. Larson. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *PVLDB*, 7(11):931–942, 2014.

[14] G. Graefe. Sorting and indexing with partitioned b-trees. In *CIDR*, 2003.

[15] G. Graefe. Fast loads and fast queries. In *Proc. DaWaK*, pages 111–124, 2009.

[16] G. Graefe. The five-minute rule 20 years later (and how flash memory changes the rules). *Commun. ACM*, 52(7):48–59, 2009.

[17] G. Graefe, W. Guy, and C. Sauer. *Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover, Second Edition*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2016.

[18] G. Graefe, M. Lillibridge, H. A. Kuno, J. Tucek, and A. C. Veitch. Controlled lock violation. In *Proc. SIGMOD*, pages 85–96, 2013.

[19] G. Graefe, H. Volos, H. Kimura, H. A. Kuno, J. Tucek, M. Lillibridge, and A. C. Veitch. In-memory performance for big data. *PVLDB*, 8(1):37–48, 2014.

[20] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record*, 26(4):63–68, 1997.

[21] J. Gray, P. R. McJones, M. W. Blasgen, B. G. Lindsay, R. A. Lorie, T. G. Price, G. R. Putzolu, and I. L. Traiger. The Recovery Manager of the System R Database Manager. *ACM Comput. Surv.*, 13(2):223–243, 1981.

[22] J. Gray and G. R. Putzolu. The 5 minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time. In *Proc. SIGMOD*, pages 395–398, 1987.

[23] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[24] T. Härder and A. Reuter. Optimization of Logging and Recovery in a Database System. In *IFIP TC-2 Working Conference on Data Base Architecture*, pages 139–156, 1979.

[25] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[26] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter. Group commit timers and high volume transaction systems. In *Prod. HPTS*, pages 301–329, 1987.

[27] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. EDBT*, pages 24–35, 2009.

[28] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Scalability of write-ahead logging on multicore and multisocket hardware. *VLDB Journal*, 21(2):239–263, 2012.

[29] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. ICDE*, pages 195–206, 2011.

[30] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. LeanStore: In-Memory Data Management Beyond Main Memory. In *Prod. ICDE*, 2018.

[31] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proc. ICDE*, pages 38–49, 2013.

[32] R. A. Lorie. Physical integrity in a large segmented database. *ACM Trans. Database Syst.*, 2(1):91–104, 1977.

[33] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *Proc. ICDE*, pages 604–615, 2014.

[34] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. EuroSys*, pages 183–196, 2012.

[35] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial

rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

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

[37] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[38] K. Park, J. Do, N. Teletia, and J. M. Patel. Aggressive buffer pool warm-up after restart in SQL Server. In *Proc. ICDE*, pages 31–38, 2016.

[39] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proc. SIGMOD*, pages 1539–1551, 2016.

[40] O. Rodeh. B-trees, shadowing, and clones. *TOS*, 3(4), 2008.

[41] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[42] C. Sauer. *Modern techniques for transaction-oriented database recovery*. PhD thesis, TU Kaiserslautern, Germany, Dr.Hut-Verlag München, 2017.

[43] C. Sauer, G. Graefe, and T. Härder. Instant restore after a media failure. In *Proc. ADBIS*, 2017.

[44] C. Sauer, L. Lersch, T. Härder, and G. Graefe. Update propagation strategies for high-performance OLTP. In *Proc. ADBIS*, 2016.

[45] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. In *Proc. SIGMOD*, pages 217–228, 2012.

[46] M. Stonebraker. The design of the POSTGRES storage system. In *Proc. VLDB*, pages 289–300, 1987.

[47] M. Stonebraker. The land sharks are on the squawk box. *Commun. ACM*, 59(2):74–83, 2016.

[48] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. SIGOPS*, pages 18–32, 2013.

[49] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proc. SIGMOD*, pages 1041–1052, 2017.

[50] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: A scalable log-structured database system in the cloud. *PVLDB*, 5(10):1004–1015, 2012.

[51] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.

[52] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *Proc. SIGMOD*, pages 1119–1134, 2016.

[53] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 323–336, 2018.