TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Modern techniques for transaction-oriented database recovery

PhD thesis

approved by the Department of Computer Science
of the University of Kaiserslautern
for the award of the doctoral degree

Doctor of Engineering (Dr.-Ing.)

to

Caetano Sauer

Date of defense:
August 21, 2017

Reviewers:
Prof. Dr.-Ing. Dr. h. c. Theo Härder (TU Kaiserslautern)
Prof. Dr. Thomas Neumann (TU Munich)
Dr. Goetz Graefe (Google Inc.)

Dean:
Prof. Dr.-Ing. Stefan Deßloch

PhD committee chair:
Prof. Dr. rer. nat. Christoph Garth

D 386

Modern techniques for transaction-oriented database recovery

CAETANO SAUER, University of Kaiserslautern

ABSTRACT

Transaction-oriented database recovery has been a "solved problem" for at least 25 years since the introduction of the ARIES methods for logging and recovery. However, recent technological developments have urged the need for new software architectures that can better exploit the efficiency of modern hardware. In the context of recovery, new methods and algorithms are required to effectively accommodate the exponential decrease in main-memory cost, the advent of flash memory, the rapid expansion into many-core CPUs, and the ever-increasing capacity of magnetic disks.

This thesis describes and evaluates a variety of new software techniques for efficient transaction-oriented database recovery. These techniques are presented in an incremental manner, starting from the widespread write-ahead logging approach of ARIES and ending with a radically new system design that relies on logging as the only form of persistent storage. The results of this thesis improve the state of the art in transaction-oriented database recovery in three major ways: (i) improved availability in the presence of failures by recovering fine-granular data items on demand; (ii) elimination of bottlenecks incurred by logging and recovery on in-memory transaction processing; and (iii) simplification of the system architecture by decoupling persistence-related components from in-memory transaction processing. The contributions herein differ from other recent research in the field by being largely hardware-independent. In particular, data is not assumed to fit entirely in main memory and the software architecture in general accommodates traditional hard disks as well as solid-state drives, non-volatile memory, and very large volatile memory.

Contents

1 INTRODUCTION

This dissertation addresses the problem of transaction-oriented database recovery, which is a fundamental technique in the implementation of the ACID paradigm of transactions [HR83]. Techniques and algorithms for database recovery have been heavily studied and proposed in over 40 years of database systems research and practice. However, these techniques must be revisited to better exploit the characteristics of modern hardware—most importantly the exponential decrease in the cost of DRAM, the emergence of fast solid-state drives, the increase in storage capacity of hard-disk drives, and the advent of multi-core CPUs.

1.1 Motivation and outline

The contributions of this dissertation are categorized in three major parts, which are presented in dedicated sections after a review of related work in Chapter 2. In the first part (Chapter 3), the problem of database system availability after a failure is addressed with a family of techniques called **instant recovery**. In the second part (Chapter 4), the general problem of **update propagation** is addressed in detail, with an in-depth analysis of different strategies, their performance and implementation, as well as the proposal of novel log-based techniques. In the third part (Chapter 5), a novel log-structured design for database persistence and recovery called **FineLine** is proposed. While this design presents a radically new approach to transaction recovery and storage management, it also arises as an extension of first two parts of the thesis.

Instant recovery addresses two fundamental problems of most existing approaches for database recovery, including the widely used ARIES methods implemented in the vast majority of commercial database systems. The first problem is that recovery is usually performed offline, meaning that the database and its applications only become available to new transactions after the full recovery process is completed. This process can take multiple hours to complete, depending on factors such as hardware characteristics, workload access patterns, and transaction volume. The second problem is that individual recovery actions, such as the replay of updates on a single data page or the rollback of a transaction, are not scheduled in a way that prioritizes the needs of applications after a failure.

Aiming to solve these two problems, instant recovery addresses a basic research and engineering challenge: to make recovery algorithms *incremental* and *on-demand*. Incremental recovery solves the first problem by enabling access to individual data pages (or contiguous sets thereof) before the complete recovery process is finished. This requires exploiting the independence of recovery among objects and reorganizing the recovery process accordingly. Building upon this independence, fine-granular recovery actions can then be scheduled following the demands of new transactions started after a failure, which essentially reduces the observed mean time to repair by multiple orders of magnitude.

The first building block of instant recovery is a technique called *single-page repair*, whose goal is to recover from hardware failures or corruptions whose effects are restricted to an individual page (or a small set of pages) of a storage device. Instead of performing media recovery on the entire device, the technique exploits per-page pointer chains in the recovery log to retrieve the history of updates of a single page. These updates can then be replayed on a backup image of the failed page to recover its most recent state.

Instant restart is a technique for recovery from system failures that builds upon single-page repair to provide incremental, page-oriented redo in addition to the traditional log-oriented redo phase of ARIES recovery. Furthermore, by collecting acquired locks during checkpoints and log analysis, the technique also enables incremental and on-demand undo actions by detecting lock conflicts

between pre-failure (i.e., "loser") and post-failure transactions. The resulting recovery algorithm allows the execution of new transactions immediately after the log analysis phase, and the access pattern of post-failure transactions actually guides the redo and undo actions required for recovery.

The remaining components of instant recovery, which are the most elaborate and therefore constituted the majority of work in this dissertation, are concerned with media failures. In a first phase of the work, a technique called *single-pass restore* was developed to eliminate random I/O operations from the redo phase of media recovery. This improves the efficiency of the recovery algorithm dramatically and, as a consequence, renders incremental backups obsolete. The new algorithm requires partially sorting the recovery log during the archiving process, so that it can be merged with a backup file in a sequential pass during recovery.

The second phase of the work on media recovery extended the single-pass restore algorithm into a technique called *instant restore*. It extends the partially sorted log archive with an index in order to allow incremental and on-demand restoration of independent segments of the database. Like instant restart, it relies on the demands of post-failure transactions to guide the recovery process and incrementally restore accessed data segments.

The second major part of this thesis addresses problems related to database checkpoints. It starts by clearly separating the concerns of checkpointing into two orthogonal aspects: *system state* and *database state*. The former type of checkpoint saves a fuzzy snapshot of the in-memory state of database system components such as the buffer pool and the transaction manager; thus, it is not concerned with database contents. Its goal is solely the reduction of the log analysis phase during restart recovery. The latter is concerned with the propagation of changes from data pages in the buffer pool to their permanent location on persistent storage. These two concerns are usually mixed in database literature, but an efficient and reusable implementation should separate them into two completely independent system services. This thesis proposes and evaluates algorithms for these services, focusing mainly on optimizing the case of large main-memory capacity and high transaction throughput.

Building upon the discussion of efficient checkpoint techniques and reusing techniques of instant recovery, this thesis presents a novel family of techniques called *decoupled persistence*. The main goal of this effort is to simplify checkpoint and recovery techniques by decoupling them from critical components of the database system such as the buffer pool and the transaction manager. The key technique employed is to rely solely on log information to perform checkpoints and propagate changes to the persistent database. This approach not only enhances the reusability of the database system's internal components, but also potentially improves performance by eliminating the interference of checkpoint actions on critical system components.

The third major part of the dissertation proposes a novel design for database storage and recovery called FineLine. Its goal is to simplify the recovery process by eliminating the persistent database, relying solely on the recovery log for data storage and retrieval. This results in a much simpler system architecture, which also decouples in-memory data structures from persistence concerns, thus improving performance for memory-resident workloads.

The principal design challenge behind FineLine is the realization of an efficient persistent data structure to maintain the recovery log. Rather than being a simple sequential file, which is optimized only for write operations, the log is partitioned and indexed in the same way as the indexed log archive used for instant restore. This index data structure, which is reorganized incrementally using techniques similar to log-structured merge trees, maintains the write efficiency of a sequential log while also providing efficient read access.

Using the novel indexed log data structure, fine-granular data items can be fetched and recovered to their most-recent, transactionally consistent state in a single transparent operation. This implies that there is no algorithmic logic that is exclusive to recovery from failures; instead, recovery is embedded in the data access protocol, without a distinction between normal and recovery processing modes.

From a more general perspective, the goal of FineLine is to provide efficient, transparent, reliable, and highly-available persistence as a decoupled component, accommodating arbitrary implementations of in-memory access methods and concurrency control. As such, it blurs the lines between in-memory and disk-based database systems.

1.2 Assumptions

This thesis builds upon a traditional database system architecture as described by Hellerstein et al. [HSH07] as well as Härder [Här05]. As the focus lies on transaction-oriented recovery following the ACID paradigm, the concepts laid out in the survey by Härder and Reuter [HR83] apply. More specifically, this work builds upon write-ahead logging techniques [Gra78] as proposed by Mohan et al. in the ARIES framework [MHL+92].

The architectural assumptions relevant for this thesis in the referred literature above can be summarized as follows. Queries and transactions of an arbitrary data model (e.g., relational) are mapped to physical operations on page-based data structures (e.g., heap-organized tables or B-trees [GR93]) accessed via a buffer pool interface. Every operation that modifies data must generate a physiological log record uniquely identified by a log sequence number (LSN). This log record contains all the information necessary to redo or undo the corresponding modification during recovery. To guide the recovery process, the buffer pool keeps track of the LSN of the latest modification that affected each page; following the write-ahead logging rule, all log records up to this LSN must be flushed before the page is written to persistent storage. Undo actions, resulting either from transaction abort or rollback of loser transactions after a failure, are logged as logical compensation actions, which guarantees "exactly once" application of log records and correctness of recovery with fine-granular (e.g., row or key value) locking [MHL+92].

The log is assumed to reside on *stable storage*, i.e., its contents are never lost, even in case of a failure. This is of course a theoretical assumption; in practice, it means that techniques for highly reliable storage—which are orthogonal to the concerns of transaction-oriented recovery—must be employed for the recovery log.

The FineLine design presented in Chapter 5 alleviates these assumptions with a more general architecture and less dependencies between internal components. Nevertheless, it is better understood, and thus also presented in this thesis, as an evolution of the traditional architecture described above.

1.3 Author's contributions

This thesis results from the collaboration of the author, Caetano Sauer, and his PhD advisor, Prof. Theo Härder, with Dr. Goetz Graefe[1] and others at Hewlett-Packard Laboratories. Dr. Graefe, who is also a co-advisor of this thesis, invented the instant recovery techniques described in Chapter 3, while the author contributed with the implementation, optimization, and evaluation of these techniques. Wey Guy provided an implementation of instant restart, which was mostly rewritten by the author but served as initial reference implementation. As a result of this collaboration, the main description

[1] Currently at Google Inc.

of instant recovery has been published in a book [GGS16], which was written primarily by Dr. Graefe. As a complement to that work, this thesis focuses on the experimental evaluation of instant recovery techniques and, where applicable, the description of implementation details.

The extensions of instant recovery and novel checkpoint techniques described in Chapter 4 are primarily a contribution of the author, but certain building blocks such as the *write elision* technique and the *partially sorted log archive* are taken from the work on instant recovery. Lucas Lersch also contributed an initial reference implementation of *log-based checkpoints* and *log-based page cleaning* in his Master's thesis; these were also rewritten by the author to a large extent. The empirical evaluation of these techniques is due entirely to the author.

Lastly, the work on *FineLine* is a contribution attributed primarily to the author. Nevertheless, Dr. Graefe is due major credit in this effort with numerous discussions and his fundamental guidance in the invention and further refinement of the design—a long process that started in early 2014 when thinking about implementing a *no-steal* mechanism for time travel and page-based snapshot isolation [SH14].

There is also significant overlap between the text in this thesis and that of other publications of the author [SGH14, SH14, SLHG16, SGH17]. These publications were also written primarily by the thesis author, and thus the overlap is not explicitly mentioned.

The concepts presented in this thesis were implemented and evaluated in the *Zero storage manager*[2], which is a fork of *Shore-MT*[3]—both of which are available as open-source repositories. These systems result from the combined effort of dozens of people in over 20 years of research. The main institutions involved in the development of Shore-MT and Zero are: University of Wisconsin-Madison, Carnegie Mellon University, École Polytechnique Fédérale de Lausanne (EPFL), Hewlett-Packard Laboratories, and, most recently, University of Kaiserslautern.

[2]https://github.com/caetanosauer/zero
[3]https://sites.google.com/site/shoremt

2 RELATED WORK

This section discusses related prior work in the domain of transaction-oriented database recovery. The focus is on traditional write-ahead logging techniques based on the ARIES family of recovery algorithms, as it constitutes the baseline upon which the work in this thesis is developed. Nevertheless, aspects of alternative recovery mechanisms—especially recent developments in research—are also briefly discussed and contrasted to the base approach of this thesis.

2.1 Failure classes

Database literature traditionally considers three classes of database failures [HR83], which are summarized in Table 1 (along with single-page failures, a fourth class to be discussed in Section 3.4). The first class—transaction failure—occurs during normal processing, when a single transaction must be rolled back due to conflicts or deadlocks in the concurrency control protocol, violation of integrity constraints, or voluntary application- or user-initiated abort. Fine-granular recovery mechanisms such as ARIES [MHL+92] and most modern recovery implementations treat transaction failures with logical compensation actions that simply revert the actions performed so far (e.g., remove an entry from an index if the original operation was an insert). As such, recovery from transaction failures actually does not typically involve the recovery component of a database system—aborted transactions are simply treated as successful transactions that did not perform any visible logical modification on the database, i.e., they "committed nothing". They may, nevertheless, perform physical changes that modify only database representation but not its contents—thereby not violating transaction consistency—by invoking *system transactions* [Gra12] (or their counterpart in ARIES, *top-level actions* [MHL+92]).

In the scope of this thesis, it is important to distinguish between system and media failures, which are conceptually quite different in their causes, effects, and recovery measures. System failures are usually caused by a software fault or power loss, and what is lost—hence what must be recovered—is the state of the server process in main memory; this typically entails recovering page images in the buffer pool (i.e., "repeating history" [MHL+92]) as well as lists of active transactions and their acquired locks, so that they can be properly aborted. The process of recovering from system failures is called *restart*.

In a media failure, a persistent storage device fails but the system might continue running, serving transactions that only touch data in the buffer pool or on other healthy devices. If the system and media failures happen simultaneously, or perhaps one as a cause of the other, their recovery processes are executed independently, and, by recovering pages in the buffer pool, the processes coordinate transparently.

The remainder of this thesis explicitly separates recovery techniques for these failure classes, employing the appropriate terms *rollback*, *restart*, *restore*, and *page repair* to describe each orthogonal recovery technique.

Table 1. Failure classes, their causes, and effects

Failure class	Loss	Typical cause	Response
Transaction	Single-transaction progress	Deadlock, constraint violation	Rollback
System	Server process (in-memory state)	Software fault, power loss	Restart
Media	Persistent database contents	Hardware fault	Restore
Single page	Local integrity	Partial writes, wear-out	Repair

2.2 Logging

> *"Perhaps the first reference to logging is found in the story of Theseus and the Minotaur. Theseus used a string to UNDO his entry into the labyrinth. Two thousand years later, the Grimm brothers reported a similar technique: Hansel and Gretel discovered that the log should be built of something more durable than bread crumbs (they reported the first log failure)."*
>
> Jim Gray and Andreas Reuter [GR93]

2.2.1 Write-ahead logging

The vast majority of database systems relies on the *write-ahead logging* (WAL) concept, which has been part of database systems folklore since the early '70s and was first documented by Gray [Gra78]. In general, the idea behind WAL is to provide transaction atomicity and durability by maintaining a separate persistent data structure, called the *log*, in addition to the *materialized database* [HR83], which is referred to here as simply the *database*. The "write-ahead" qualifier refers to the ordering constraint imposed on writes performed on the log and on the database: all log records pertaining to a given data item must be made durable before that data item is updated on the database[4].

From a general perspective, the opposite of WAL is any design that maintains a single persistent storage structure. Traditionally, this single storage structure has been the database, in designs that in some form or another resemble the *shadow paging* approach [Lor77]. These include, for instance, traditional designs such as the original scheme developed by Lorie [Lor77], Härder and Reuter's TOSP approach [HR79], and Severance and Lohman's differential file mechanism [SL76]. Modern incarnations of this technique include the hardware-supported isolation mechanism of Hyper [KN11] as well as atomic copy-on-write B-trees used for key-value stores and file systems [Rod08, Chu17, RBM13].

Shadow paging approaches have three major drawbacks. First, they are inefficient with conventional storage devices such as magnetic disks and solid-state drives, since the absence of a log requires the use of an *atomic* propagation strategy with *no-steal* caching and commit processing with a *force* policy [HR83]. This configuration can only perform reasonably well with large main-memory capacity and storage latency very close to that of main memory. Second, they severely limit concurrency by requiring either page-level locking [HR79, Lor77, GMB+81] or a single-writer execution model [KN11, Chu17]. Third, they tend to be less useful and less reliable because of the lack of history. The log is a useful tool for debugging, auditing, accountability, and time-travel queries. Furthermore, due to its non-destructive write pattern, the log is more effective in supporting a wide range of failures, including media and single-page failures.

A less obvious alternative to WAL designs is to eliminate the materialized database and rely on the log as the only form of persistent storage. Chapter 5 proposes such an approach and discusses related work in that domain. Other non-WAL approaches proposed for non-volatile memory are discussed in Section 2.7 below.

[4]The term "ahead" can be confusing because it implies different orderings depending on the interpretation. The word can be defined as "forward in space", as in "take the road exit ahead", or "in the lead", as in "Michael Phelps finished eight seconds ahead". Note that this latter definition is the opposite of "forward in time". Thus, "write ahead logging" may imply a different temporal ordering constraint than "write-ahead logging"—does the write "come first", like Michael Phelps, or does it "come later", like the road exit? A commonly overlooked hyphen makes all the difference here. To clarify this confusion while still maintaining the famous acronym, a better name would be "write after logging".

The vast majority of database systems today implement write-ahead logging as described by Mohan et al. in the ARIES framework [MHL+92]. The two key distinctions to its predecessors are the use of *physiological logging* and *logical undo with compensation*. The former enables "recovery independence amongst objects" [MHL+92], a key characteristic exploited on all techniques presented in this thesis. In essence, each log record describes changes on a given set of database pages, and a redo operation is guaranteed to have its effects restricted to those pages only. This implies that redo actions on different pages can be performed independently. The latter technique—logical undo with compensation—dictates that undo actions must be the logical compensation of the original action; "logical" here implies granularity greater than a page, i.e., a table, index, or the whole database. For example, an insertion of a record on a table is undone as the deletion of that same record from that table, regardless of the effects on physical data structures—during undo, the record might have been moved to a different page due to a page split operation. These undo actions generate special redo-only log records known as *compensation log records* (CLR), which are processed like any other log record during redo but guide the undo logic in a way that guarantees *idempotence*, i.e., recovery actions are applied exactly once, even in the presence of system failures. The key advantage of this undo scheme is that it enables partial rollbacks, and, most importantly, *record-level locking*, i.e., concurrency control with granularity finer than a page, as proposed, for example, in ARIES/KVL [Moh90a] and ARIES/IM [ML92].

2.2.2 Log contents

In a general sense, the log contains two types of log records: log records generated by transactional updates and log records generated by system activities. The distinction between these two types will become clear in the discussion of checkpoints in Section 2.3 below, as it pertains to an important discrimination between *database state* and *system state*. For now, we focus on the contents of log records generated by transactional updates.

Figure1 shows the organization of log records within a log file and their internal structure. The *header* and *footer* parts of a log record contain all metadata which is relevant for checkpoint and recovery procedures. The example in the diagram considers only metadata fields in the header, but additional information may be stored or even replicated in the footer for consistency checks (e.g., a checksum or a copy of some header fields). The `type` field specifies what kind of action is logged and, consequently, what procedures must be invoked to redo or undo it. The log sequence number (LSN), stored in the `lsn` field, uniquely identifies each log record and serves two main purposes: it guides recovery, checkpoints, and log recycling by storing on each database page the LSN of the last action that modified it; and second, it can be used to locate a log record in a log file with direct addressing. A typical implementation uses a pair consisting of log file number and byte offset within that file. The field `length` stores the length of the log record in bytes; this is required because log records have variable size.

The following header fields, shown in blue in the diagram, pertain to undo processing. The `txn_id` field contains the identifier of the transaction that performed that action; it is used during checkpoints and recovery to determine what transactions are still active and may thus require rollback during restart. The `store_id` field specifies the logical identifier of the table, index, or database—in general, the *store*—in which the action was performed. As discussed earlier, undo processing must perform logical actions at this granularity, and thus the logical identifier is required here. The `txn_prev` field is a pointer to the previous log record generated for that same transaction. The `undo_nxt` field is generated for CLRs to guide the undo process in case of a rollback action interrupted by a system failure; it guarantees idempotence of undo actions. When undoing a particular action, the

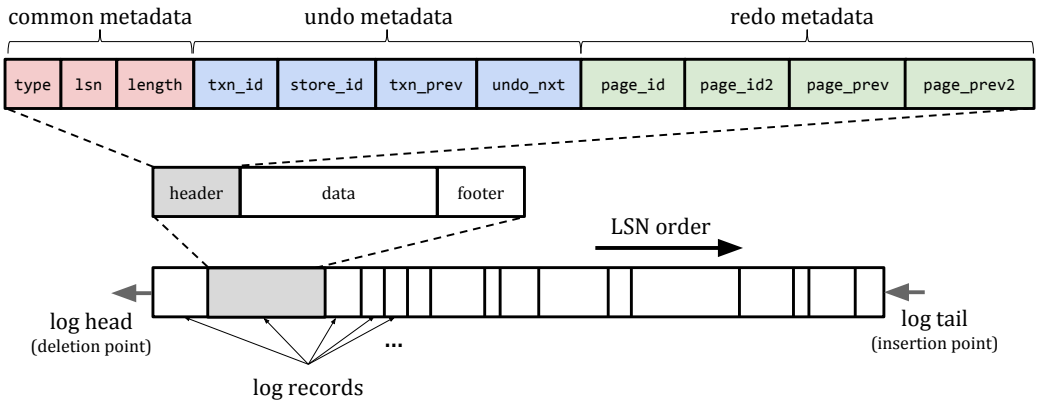

Fig. 1. Log file contents and log record header structure

undo_nxt field of the generated CLR contains the txn_prev field of that action's log record; this allows rollback processing to skip undoing that action a second time.

Simple optimizations proposed by Graefe et al. [GGS16] allow the elimination of both the undo_nxt and the txn_prev fields by maintaining a list of LSN pointers to all log records of each active transaction in an in-memory data structure.

The following header fields, shown in green in the diagram, pertain to redo processing. The page_id field contains the identifier of the main page that was affected by that action—this is the key component of the physiological logging approach. As an optimization, a second field page_id2 can be used for single-log-record system transactions [Gra12] that move data from a source page to a destination page, e.g., a B-tree node rebalance following a split or merge [GKK12]. This mechanism can be generalized to an arbitrary number of pages per log record, but in practice two is ideal as it supports most useful operations with simples logic than the arbitrary case. The page_prev field points to the previous log record whose action affected the page with identifier page_id; the same holds for page_prev2 and page_id2.

The maintenance of a *per-page log chain* with page_prv fields is not mentioned in the original ARIES publication [MHL+92], but it has been used in practice by systems like Oracle—to support snapshot isolation or "read consistency" using undo log records [M+04]—and SQLServer—to provide consistency checks during recovery and log shipping[5]. The per-page log chain is an essential building block of the techniques discussed in this chapter, as it enables the retrieval of log records pertaining to a given page without scanning the log.

Figure 2 shows an example of log file fragment containing six log records; the per-page and per-transaction log chains are illustrated on the left and right sides of the diagram, respectively. The example shows two transactions, with identifiers 1 and 2, performing updates on pages A, B, and C that are nodes of a Foster B-tree data structure [GKK12]. In this diagram, the page_id and page_id2 are unified into a single field for ease of exposure; the same holds for page_prev and page_prev2.

[5]Personal communication from Goetz Graefe

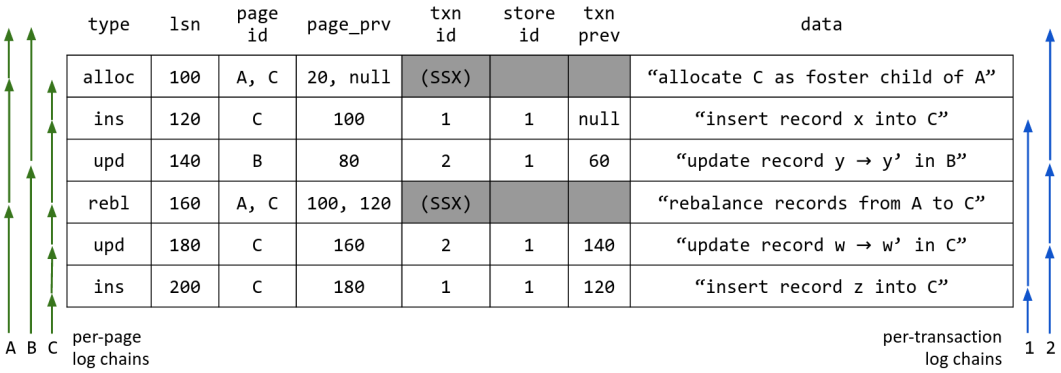

Fig. 2. Example of log records with per-page and per-transaction chains

The first log record, of type `alloc`, allocates a new, empty B-tree node C and assigns that node as a foster child of A. This operation is a single-log-record system transaction (SSX), which produces a single redo-only log record; the mere presence of such log record indicates the successful commit of the transaction [Gra12]. For these reasons, undo information can be omitted. The log chain of A continues on to the log record with LSN 20, which is omitted here, while C points to `null`, indicating that the chain shall terminate here.

Next, transaction 1 inserts a new record `x` into the newly created page C; thus, it points to the previous SSX that first allocated C. As this is the first log record of this transaction, its `txn_prev` field is set to `null`. In the next log record, a transaction 2, which previously made an update with LSN 60, as the per-transaction log chain shows, updates a record in page B. After that, another SSX comes in and rebalances the records of nodes A and C; in the Foster B-tree data structure, this is part of a multiple-step node split operation [GKK12]. Note that because these are independent system transactions, transaction 1 was able to insert a record into C before the rebalance operation. The per-page log chains of A and C are adjusted accordingly.

Finally, transactions 2 and 1 modify page C, and the final state of the log chains is shown with the arrows on the sides of the diagram.

2.2.3 *Log buffer and commit processing*

The discussion of logging techniques so far has focused only on the persistent log. However, the path from creating a log record in main memory prior to a transactional update and the final, durable propagation of that log record into the persistent log is a critical aspect of scalable system design. In order to make this critical operation as efficient as possible, it is crucial to provide a scalable, low-overhead log buffer implementation. As portions of the log buffer are made durable, transactions waiting for the commit acknowledgement are notified; thus, the log buffer implementation is tightly tied to commit processing.

Figure 3 illustrates the log buffer and commit processing for ARIES-based logging. In this example, three transactions, T_1, T_2, and T_3, are currently active and producing log records for each of their transactional updates, represented here by black dots. Each log record is inserted into the log buffer sequentially, and LSN values are assigned depending on the order of insertion. In this diagram, log records are represented as rectangles inside the log buffer and colored according to the transaction that generated them. Under the log buffer, the three arrows denote three ranges of interest in the log buffer. First, the *already flushed* range has been safely made durable, meaning that its space can

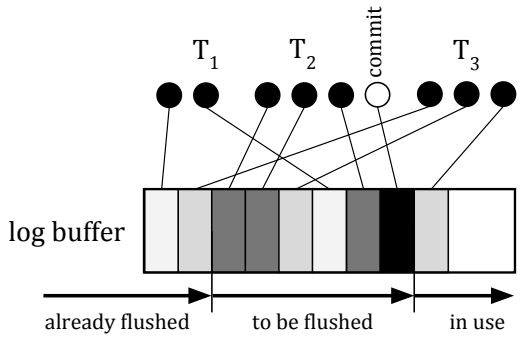

Fig. 3. Log buffer and commit processing in ARIES

be freed in the log buffer (typically implemented as a contiguous circular buffer in main memory). The range *to be flushed* contains the log records that have been successfully inserted into the log buffer (and thus are immutable at this point) but are still waiting to be persisted. Finally, the tail of the log buffer is the *in use* range; here, log records are still being inserted or manipulated by the system in some way, so that their contents are not yet final. In the example, the last log record of T_3 is still being generated.

The white dot in the diagram represents a commit request from transaction T_2. It generates a commit log record (shown as a black rectangle) in the log buffer, and the commit can only be acknowledged once it is flushed, i.e., once it enters the *already flushed* range. One crucial requirement of this logging scheme is that all operations are strictly ordered—a commit can only be acknowledged once a commit log record and *all* preceding log records have been safely generated, assigned an LSN value, and flushed to persistent storage. Another way to look at this rule is that the log should not have any "holes".

The state of the art in log buffer and commit processing techniques for ARIES-based logging is the Aether approach of Johnson et al. [JPS+12]. Aether provides a more scalable implementation of the *insert* operation of the log buffer with two techniques called *consolidated allocation* and *decoupled buffer fill*. The former allows threads performing insertion to consolidate the allocation of space in the log buffer, so that the thread currently in the critical section can allocate space on behalf of multiple threads. These threads form a *consolidation group*, which is built from threads waiting to join the critical section. The latter technique allows threads to fill their allocated buffer space in parallel; thus, the only operation that must be serialized is the allocation of log buffer space, and not the actual copy of data into the allocated space. Additional synchronization is then needed in the *release* phase (i.e., after a thread has allocated and filled its log buffer space) to ensure durability and avoid "holes" in the log in case of a system failure. To that end, the insertion of a log record is only acknowledged (i.e., the insert call only returns) once all previous space has been properly filled. Aether also provides the option to perform this synchronization in a decoupled manner, using a *delegated release queue*, in which, similarly to consolidated allocation, one thread can release space on behalf of others.

Besides a more scalable log insert operation, Aether also decouples the system call and I/O overhead of flushing log records to persistent storage with a technique called *flush pipelining*. Rather than have each thread perform not only insertion but also flush the log buffer at commit time, a dedicated *flusher* thread keeps track of the current range of the log buffer that has been properly

released after insertion but not yet flushed. This range is then periodically flushed following a certain policy. Thus, commit processing involves (i) inserting a commit log record, (ii) requesting a flush on the flusher thread, and (iii) waiting until the LSN immediately after the commit log record is made persistent by the flusher thread. This technique has several advantages: first, it drastically reduces operating-system scheduling overhead that results from multiple threads performing *write* system calls [JPS+12]; second, it naturally provides *group commit* capabilities [DKO+84, GK85, HSL+87] with flexible flush policies; and third, it is also a natural fit for *early lock release* [DKO+84, GLK+13], since worker threads can be reused immediately after generating the commit log record to perform work of other transactions.

The ARIES-based logging technique discussed so far requires establishing a total order on log records produced by all concurrent threads executing transactional updates. In multi-threaded scenarios, this single point of synchronization can become a scalability bottleneck. Wang and Johnson [WJ14] have extended the Aether approach with distributed logging in NUMA architectures with non-volatile memory. As discussed in detail in their work, ARIES-based recovery can be implemented by maintaining a partial order on log records. As long as log records pertaining to the same page are ordered for redo processing and log records pertaining to the same transaction are ordered for undo processing, correct transaction recovery can be guaranteed. To that end, Wang and Johnson propose a generalized LSN mechanism based on Lamport's logical clocks [Lam78]. This mechanism enables the use of multiple log buffers, partitioned either by transaction or page identifier.

A much simpler way to implement distributed logging is to employ a *no-steal* eviction policy. In this case, the lack of undo recovery implies that log records need not be ordered according to transaction identifiers—a single order on page identifiers suffices for correct redo recovery. The Silo in-memory database implements a simple distributed logging protocol based on this idea [TZK+13, ZTKL14]. Chapters 4 and 5 of this thesis present *no-steal* designs that support distributed logging in the same way. Note that despite being developed for a centralized logging scheme, the principles behind the Aether log manager [JPS+12] still apply in a distributed scenario to make each individual log buffer more scalable, thereby reducing the total number of distributed logs.

2.2.4 Log storage

One last important aspect of logging that is worth mentioning is storage management. Technical details of log implementation and practical issues that must be addressed are covered in depth by Gray and Reuter [GR93]. This section briefly summarizes some of the key issues as a motivation for simpler techniques presented later in Chapters 4 and 5.

Conceptually, the log is a circular data structure, which, as shown in Figure 1, is only modified by appending to the tail and removing from the head to recycle space. Recycling (also referred to as truncation) is usually performed at some coarse granularity such as a file or a "virtual log" [Mic17b]. Determining the point up to which the log can be recycled is one of the two major tasks of checkpoints, which are discussed in Section 2.3 below.

When a system failure occurs, the log must be brought to an internally consistent state; this process is called *log restart*. The key issue is determining the current *durable LSN*, i.e., the last log record to have been fully persisted without any preceding "holes". Some log records may be lost due to interrupted writes, but these do not incur loss of transactional durability thanks to the commit processing mechanism discussed earlier for the log buffer. To guarantee such durability, it is crucial to organize log files into pages that are appended once but never overwritten; this implies that

storing log records contiguously in a file does not guarantee durability, because the file system must overwrite its own pages to maintain contiguousness[6].

Lastly, the log manager must provide a *reservation* scheme to guarantee the ability to roll back any transaction. When the log is close to its full capacity, the system may get "stuck" if it cannot roll back a transaction because there is no free log space for the required CLRs. This critical situation is further complicated by the fact that reservation is "fuzzy": a transaction usually does not know precisely how much log space it requires to roll back, so it must overestimate this by a large margin when reserving log space (Section 9.6.1 of Gray and Reuter's book [GR93] discusses this problem in detail). A similar problem may occur if a rollback requires a node split but the database device is full [GGS16]. These issues can be solved if CLRs are fixed-length log records that point to the original update and if all space-management tasks are performed with system transactions [Gra12, GGS16]. More importantly for the context of this thesis, these problems illustrate the complexities involved in traditional ARIES-based logging, especially to support the *steal* eviction policy [HR83] and, by consequence, undo recovery.

2.3 Checkpoints

As a general classification, a checkpoint is any provision employed by the system during normal processing to (i) reduce the costs of recovery in case of a failure and (ii) reduce the amount of log that must be kept to ensure atomicity and durability of transactions. While not strictly required for correct transaction-oriented recovery, checkpoints are an absolute requirement in practice, as otherwise the log may grow indefinitely and the time to recover from a system failure may become directly proportional to the mean time to failure [HR83]. This section briefly reviews checkpointing techniques for ARIES-based recovery, emphasizing aspects that are relevant for the remainder of this thesis.

2.3.1 Checkpoint basics

Checkpoint techniques can be classified by the kind of consistency they guarantee on the database after a failure, by the kind of information they save, and by the kind of interference they incur in normal transaction processing.

The type of checkpoints employed in ARIES and in the vast majority of current database systems is the *fuzzy checkpoint* as first proposed by Gray [Gra78] and Lindsay et al. [LSG+79]. The term "fuzzy" is used to indicate that whatever information is captured does not reflect an exact state of the system at a particular point in time; rather, information is collected as the state changes. The main goal of fuzzy checkpoints is to minimize interference on running transactions, allowing checkpoints to be produced without "quiescing" transactions. While there is no commonly accepted definition of fuzzy checkpoints in the literature, this seems to be the most useful. Härder and Reuter present fuzzy checkpoints as a fourth type of checkpoint—an alternative to transaction-oriented, transaction-consistent, and action-consistent checkpoints [HR83]. However, according to the definition above, fuzziness is an orthogonal dimension; an action-consistent checkpoint, for instance, can be fuzzy if transactions are only quiesced at a page level, i.e., if checkpoints acquire shared latches on pages. This is actually the type of checkpoint employed in ARIES [MHL+92].

Although checkpoint information can be saved in any kind of safe storage device, log records specifying the logical "begin" and "end" times of a checkpoint operation must be saved in the log. These allow the recovery procedure to establish the validity of a checkpoint and what portion of

[6] The Shore-MT [JPH+09] log manager used in this thesis suffers from this problem.

the database history was covered by it. Given that transactions cannot be stopped to produce a checkpoint, the checkpointed information is guaranteed to be correct only for updates performed before the "begin" log record. If a system failure occurs in the middle of a checkpoint procedure, the lack of an "end" log record indicates that whatever information was persisted by that checkpoint must be ignored.

In order to support undo recovery, checkpoints must save a list of currently active transactions and the LSN of their oldest log record; this establishes a *low-water mark* [GR93] for undo recovery, i.e., the lowest LSN that may still be required to rollback all loser transactions in case of a system failure. This value can only increase when user transactions commit, and thus no system activity can proactively reduce the span of undo recovery.

For redo recovery, checkpoints are used not only to save a low-water mark (i.e., the lowest LSN that may be required to redo all changes not yet reflected in the database), but also to proactively increase it by flushing *dirty* pages to the database. A dirty page is any page whose contents have been updated in the buffer pool but not yet reflected in the persistent database. Early checkpoint approaches involved flushing all dirty pages, so that the "begin" LSN of the latest complete checkpoint was used to establish the low-water mark. Indirect checkpoints [HR83], on the other hand, save information about dirty pages currently in the buffer pool. To support indirect checkpoints, the buffer pool must keep track of the *recovery LSN*, or *dirty LSN*, of each dirty page. This is the LSN of the update that caused the page to go from a clean to a dirty state, i.e., the LSN of the first update not yet reflected in the persistent version of that database page. The redo low-water mark is then simply the minimum recovery LSN of all dirty pages collected during the checkpoint.

Note that indirect checkpoints can be combined with direct checkpoints, i.e., page contents can be flushed in addition to the dirty page table. This is the mechanism proposed by Lindsay et al. [LSG+79]. However, as argued in Section 2.3.2 below, these concerns should be separated. Furthermore, despite this being uncommon in practice, an indirect checkpoint may not be fuzzy, i.e., collecting the dirty page table may require quiescing the system. Thus, for classification purposes, fuzziness and indirection can be seen as orthogonal dimensions.

The *restart low-water mark* is defined as the minimum between the redo low-water mark, the undo low-water mark, and the "begin" log record of the latest complete checkpoint. For restart purposes, only the log records from this low-water mark onwards are required for correct recovery, and thus the log can be recycled up to this mark. Note that this says nothing about media recovery, for which an additional *restore low-water mark*, to be discussed in Section 2.8, must be considered.

2.3.2 System vs. database state

So far, checkpoints have been discussed in a general sense, but from a system-architecture perspective, it is useful to distinguish the information being saved in a checkpoint into two categories: *system state* and *database state*. The list of active transactions, for instance, reflects system state; it is a (fuzzy) snapshot of the system's transaction manager and does not contain any user data, i.e., data copied from database pages. The same holds for the list of dirty pages and their recovery LSNs. However, if a checkpoint is used to also flush dirty pages, then it is directly saving database state. Approaches that perform all of these actions (e.g., flushing pages that have been dirty since the last checkpoint [LSG+79]) are concerned with both system and database state.

As discussed in previous work [SGH14], delegating database state propagation to a dedicated *page cleaner* service and have checkpoints consider only system state achieves greater separation of concerns in the system architecture. This is especially useful if recovery starts with a log analysis phase, as in ARIES and instant restart. In that case, because capturing system state does not require

any I/O, it can be done relatively cheaply and thus at a higher frequency. This, in turn, directly reduces the length of the log analysis phase.

The page cleaner, on the other hand, is heavily I/O-bound. Thus, it usually runs independently in the background, making constant, incremental progress rather than performing all required steps in short, frequent bursts of activity. For recovery purposes, its sole function is the reduction of redo recovery costs, by increasing the redo low-water mark and reducing the number of dirty pages in the buffer pool. Chapter 4 of this thesis discusses and evaluates page cleaning techniques in detail.

From this point on in this thesis, the term "checkpoint" will be used to refer exclusively to the action of saving system state. When referring to database state, the term "page cleaning" will be used.

2.3.3 In-memory databases

As discussed so far, checkpoint techniques are hard to classify and sometimes the same term can be applied with different meanings throughout the literature. The recent popularity of in-memory databases in the research community worsens the problem because a checkpoint is usually conceptually the same as the materialized database [HR83] for recovery purposes[7]. Therefore, the distinctions between fuzzy and non-fuzzy, direct and indirect, system and database state, action- and transaction-consistency, etc., become even more blurred. Nevertheless, the present section attempts to lay some common ground and identify useful guiding principles for efficient and reliable data management.

What most in-memory database system designs call a checkpoint procedure is in fact page (or record) cleaning with an *atomic propagation scheme* and a *no-steal* eviction policy [HR83]. Furthermore, as most atomic propagation schemes, they resemble the shadow paging approach [Lor77] in many ways. The first version of Hyper [KN11], for instance, relies on a serialized execution model where long-running, read-only transactions issue a *fork* system call to read data concurrently in an isolated way. The forked child process essentially reads shadow versions of all pages it accesses, undoing updates of active transactions if necessary. However, unlike the traditional approach [Lor77], the on-demand mapping and copying of pages is implemented by the operating system with hardware support. Using this mechanism, a checkpoint is nothing but a read-only transaction that dumps the whole database into persistent storage.

For checkpoint techniques in other in-memory databases, the only distinction to the shadow paging scheme is simply that shadows are created and managed at the record granularity rather than page granularity. This addresses one of the main disadvantages of shadow paging: the requirement for page-level locking—if shadow records are used, record-level locking suffices. H-Store [MWMS14], for example, enters a special copy-on-write mode when checkpoints are being taken. In this mode, transactions must create a new version of every tuple they modify. After a checkpoint is taken, the system goes back into its normal in-place update mode. The main disadvantage of this approach is that the system runs much slower than normal when a checkpoint is being taken, i.e., checkpoints interfere too much with normal processing. This discourages frequent checkpoints and thus inevitably increases mean time to repair.

The wait-free zig-zag and wait-free ping-pong techniques [CSS+11] address the interference problem by maintaining either one or two full additional copies of the complete database. The designs very closely resemble Lorie's design [Lor77], with indirect allocation bitmaps and shadow versions of records. The main difference is that copies of the whole database are maintained at all times to reduce the overhead of creating record copies on demand, as in H-Store. On the other hand,

[7] Note how the discrimination of certain database systems as "in-memory" is quite questionable from this perspective.

significant overhead is added to reads and writes through the indirections, bitmap maintenance, and duplicated copies. Given the substantial memory requirements, these mechanisms are tailored to small databases—the paper uses multi-player online games as main target application [CSS+11]. Lastly, the idea of doubling storage consumption to optimize recovery resembles the TWIST approach of Reuter [Reu80].

What all approaches mentioned so far have in common is that the shadowing technique they employ to produce fuzzy checkpoints with little overhead is essentially a form of multi-version concurrency control [WV02]. This leads to a key insight into how checkpoints are implemented in in-memory databases: mechanisms for isolation of read-only transactions can be reused to create transaction-consistent checkpoints. The H-Store copy-on-write technique, for instance, can easily be applied to execute long, read-only transactions (e.g., OLAP) in isolation from the single writer in each partition.

The CALC approach [RDAT16a] combines the low interference of wait-free ping-pong [CSS+11] and the low memory consumption of H-Store [MWMS14] by establishing *virtual points of consistency*. The approach relies on shadow versions of records and a bitmap that controls how shadows are updated. The key technique used here is to organize the checkpoint process in multiple phases and ensure that transaction boundaries do not spawn more than two phases—before transitioning from one phase to the next, the system waits until every active transaction that started in the previous phase is finished. The way that shadows and the bitmap are updated during both transactional updates and commit processing depends on the phase observed at transaction begin. The actual write of records to persistent storage is performed by an asynchronous thread in a specific phase. During this phase, all shadows read by the writer thread are guaranteed to contain the transaction-consistent before-images at the virtual point of consistency established in the first phase of the algorithm. The correctness of the technique relies not only on shadows and bitmaps, but also on correct transaction ordering with begin and end timestamps and isolation with record-level locking.

Despite the sophistication of these low-overhead, shadow-record-based approaches, all of them incur significant interference on running transactions, so that transaction throughput unavoidably drops and transaction latency unavoidably increases during checkpoints [RDAT16a]. ARIES-based page cleaning, on the other hand, is barely noticeable, and it can even run constantly in the background, delivering robust transaction throughput. In ARIES, propagation is managed at the page level, and there is no need for shadow copies or bitmaps of any kind. The only overhead required to clean a page is to latch it in shared mode for the short duration of an in-memory copy operation of the page's contents into a separate page-cleaning buffer (details in Chapter 4). Furthermore, contention on frequently-updated pages can be minimized by either using pessimistic latches in conditional mode (i.e., only acquire the latch if it's not currently held in exclusive mode) or by using optimistic latches [LSKN16]. Because the database is kept in an action-consistent state, skipping a hot-spot dirty page is also perfectly fine—it does not invalidate the consistency of persistent storage in any way.

Despite all its advantages, modern checkpointing techniques for in-memory databases argue that the main disadvantage of ARIES—or, more precisely, of action-consistent materialized databases and physiological logging—is the overhead of logging for memory-resident workloads, especially for scalability reasons. The techniques proposed in Chapters 4 and Chapter 5 aim to eliminate this overhead with novel designs incrementally evolved from ARIES.

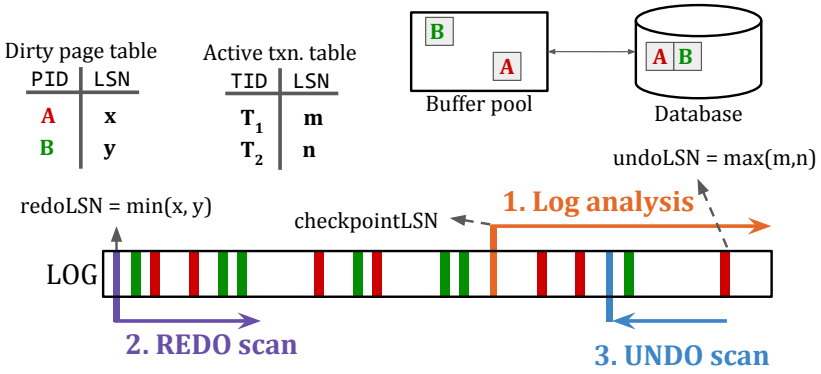

Fig. 4. System state and three log scans of ARIES restart

2.4 ARIES restart

Restart after a system failure in ARIES relies on three phases, each of which performing a log scan: log analysis, redo pass, and undo pass. The process has been discussed extensively in the literature [MHL+92], including a previous publication on instant recovery [GGS16]. Therefore, this section provides a brief overview of the restart procedure in ARIES, focusing on the factors that determine the cost and total length of each phase.

Figure 4 illustrates the three phases of ARIES restart. The remainder of this section refers back to it to illustrate each phase in detail. The example shows two pages of interest, A and B, in red and green colors, respectively. The log records affecting these pages are shown in the log below as stripes with the corresponding colors.

2.4.1 Log analysis

The first phase of ARIES restart, log analysis, is concerned solely with system state as defined in Section 2.3.2. It scans the log from the "begin" point of the latest complete checkpoint—the location of which is maintained in a pre-determined, safe storage location called the *master record*. The goal is to collect information about pages that were dirty and transactions that were active (i.e., loser transactions) at the time of failure; this is essentially the information needed do perform redo and undo recovery, respectively. Log analysis collects this information by loading the data produced by the indirect checkpoint and updating it as it scans the log forward. One crucial aspect of this phase is that it does not perform any I/O on the database, and the log scans are purely sequential.

Using the recovery LSNs in the dirty page table, the redo low-water mark is computed and passed to the next phase—the redo pass. Likewise, the minimum begin LSN of loser transactions is passed as argument to the undo pass later on. In the diagram of Figure 4, the checkpoint begin is found in the LSN *checkpointLSN*, and log analysis scans the log forward from that point until the log tail. Pages A and B were determined dirty during log analysis, with recovery LSNs x and y, respectively; similarly, transactions T_1 and T_2 are the loser transactions with begin LSNs w and z, respectively. During this process, the buffer pool remains empty.

The length of the log scan in the analysis phase depends solely on the frequency of checkpoints. Since, as discussed earlier, checkpoints are concerned solely with system state, they can be taken very frequently—for example, every few seconds. The duration of log analysis is approximately equal to or shorter than the checkpoint frequency, assuming that the write bandwidth of the log is

not greater than its read bandwidth. This is because the maximum amount of log that must be read is roughly equal to the amount of log that was written since the last checkpoint.

2.4.2 Redo pass

The redo log scan starts from the minimum recovery LSN in the dirty page table (*redoLSN* in Figure 4). Unlike log analysis, this phase usually spans a much larger range of the log and is very costly because it reads pages from the database as their log records are encountered for the first time. Also unlike log analysis, this phase is concerned solely with database contents. A previous empirical analysis found this phase to be by far the longest of ARIES restart [SGH14], especially in write-intensive OLTP workloads with short transactions, like TPC-C.

Following the *repeating history* requirement, updates of both winner and loser transactions must be replayed [MHL+92]. This includes compensation log records discussed in Section 2.2. The redo action of a scanned log record is only applied if its LSN is greater than the page LSN value found in the database.

The number of random database reads performed during redo is precisely the number of entries in the dirty page table. This is because the number of dirty pages at any point in time is at most the total number of frames in the buffer pool. Therefore, page replacement should not occur during redo and each page should be fetched only once. This assumes that the buffer pool capacity was not reduced from what it was before the system failure. If the system is restarted with less main memory for some reason, more random I/Os might be required.

As discussed earlier in Section 2.3, the redo low-water mark, and therefore the length of the redo log scan, depends solely on page cleaner activity before the failure. If write bandwidth is well-balanced and the page cleaner runs constantly in the background, the redo scan is expected to be very short. However, this is hardly the most common case in practice, and such configuration is not the most cost-effective, especially with the techniques proposed in the next chapters of this thesis. One major disadvantage of ARIES restart, at least in its initial description, is that new (i.e., post-failure) transactions cannot be executed before this costly phase is finished.

2.4.3 Undo pass

Unlike the previous phases, the undo pass scans the log backwards, from the log tail to the undo low-water mark. It starts on the largest log record produced by an active transaction (*undoLSN* in Figure 4) and finishes when all loser transactions have been rolled back. Thus, the log scan naturally stops at the undo low-water mark, without the need to explicitly compute it. Because the redo phase populates the buffer pool and their updates are redone while repeating history, undo actions should not incur database reads, unless long-running transactions, whose updates reach beyond the redo low-water mark, must be rolled back. Also unlike the previous phases, the undo pass does not scan the log sequentially; rather, it keeps track of the next log record to be undone for all active transactions. As each undo action is applied, the entry of the corresponding transaction is updated and the next log record is recomputed. If a CLR is found, the `undo_nxt` pointer is traversed to find the next log record to be undone.

The cost and length of the undo pass depends solely on workload characteristics. Typically, it is very short—in some cases even shorter than log analysis. However, in workloads that have long-running read-write transactions, running on with well-balanced write throughput, the undo pass might be by far the longest. Unfortunately, as discussed earlier, system-initiated provisions such as checkpoints cannot shorten this phase. This is another major disadvantage of ARIES recovery: a single, rarely-executed, long transaction may delay the processing of new transactions by several

minutes or even hours, even if it did not touch any data in the application working set. This makes the total time required for ARIES restart, and thus the time the system remains unavailable after a system failure, quite unpredictable.

2.5 Restart optimizations

Attempts have been made to optimize ARIES restart in three different directions: (i) reducing the amount of recovery work; (ii) performing actual recovery tasks more efficiently by parallelizing redo and undo actions; and (iii) enabling access to the database before recovery is completed. These attempts are briefly summarized below, focusing on their advantages and disadvantages.

2.5.1 Logging page flushes

The log analysis phase loads the dirty page table from the latest checkpoint and scans the log forward, adding new dirty pages as it finds updates to pages not yet in the table. However, it never removes a page from the table, because when a page is cleaned, it leaves no information in the log. If whenever a page is marked as clean in the buffer pool, a *page-flush log record* is generated, then log analysis has the opportunity to remove that page from the dirty page table. If that page was not dirtied again until the system failure happened, the redo pass will perform one less random page read.

The original ARIES publication mentions this optimization [MHL+92], but advises against it due to the increased log volume. However, modern database systems with high transaction throughput can greatly benefit from this technique, given the fast pace at which pages are cleaned and dirtied. Furthermore, the space occupied by such log records is very small in comparison with the log volume produced by transactions.

A previous study on this issue [SGH14] revealed that, depending on the exact point in time a system failure happens, up to 25% of the entries in the dirty page table can be false positives (i.e., pages fetched from the database during redo recovery that do not actually require any recovery). Thus, logging page flushes is a very useful technique to reduce recovery costs.

2.5.2 Parallel redo

The key advantage of the physiological approach is that it provides independence of redo recovery among pages. This means that redo log records pertaining to any given page can be applied independently of the state of other pages. For example, updates on a leaf page of an index can be reapplied before any update is reapplied on its parent, even if the original actions were applied first on the parent. As another example, records can be modified on any table without catalog pages being recovered first.

While this independence can be very powerful if fully exploited during recovery, proposed optimizations still rely heavily on the redo log scan to determine the order in which redo actions are applied, without any regard for what pages the application might need first after a failure. For instance, Mohan et al. suggest building "in-memory queues of log records which potentially need to be reapplied" [MHL+92]. This technique can be combined with asynchronous reads of pages in the dirty page table, so that each queue is reapplied as the corresponding page becomes available. The problem with this technique is that it still uses the redo log scan as the primary scheduler of redo actions, simply exploiting asynchronous reads to reduce stalls, i.e., avoid blocking the redo scan while a page is fetched from the database.

The C-ARIES approach [SK07] exploits redo independence among pages to a greater degree by performing redo in a page-by-page manner. Using a modified log analysis phase, the algorithm

collects not only a list of dirty pages but also a linked list of log records for each page—this list contains all log records pertaining to that page from its recovery LSN up to the its latest update. This modified analysis phase requires scanning the log from the restart low-water mark rather than from the latest checkpoint. Because this mark is only advanced as pages get cleaned and the oldest transactions commit, the length of the log analysis scan can be multiple orders of magnitude larger than in traditional ARIES.

During the redo phase, a thread is spawn for each dirty page and the corresponding log records are applied in parallel. Since the number of dirty pages can easily reach hundreds of thousands [SGH14], this type of parallel redo causes extreme saturation on the thread scheduler of the operating system. A better approach would be to introduce some kind of priority scheduling for page-by-page redo, preferably accounting for the needs of post-failure transactions.

2.5.3 Parallel undo

While physiological logging enables redo independence among pages, logical undo in ARIES enables undo independence among transactions. This means that, in principle, loser transactions can be rolled back in any interleaved schedule. A potential concern could be transaction isolation, but two-phase locking during normal processing guarantees that actions performed by active transactions do not conflict with each other; therefore, the rollback of those actions is also free of conflicts, because locks are not released until the transaction commits. Mohan et al. suggest performing undo in parallel by assigning the rollback of each loser transaction to an independent thread [MHL+92].

Despite the possibility of transaction-oriented undo, recovery algorithms implemented in practice—such as Shore [CDF+94]—perform log-oriented undo in strict reverse chronological order instead. The undo pass algorithm presented by Mohan et al. also uses reverse chronological order [MHL+92]. This is likely due to issues that arrive with free-space management during logical undo; for example, undoing a record deletion might require splitting a page to perform the compensating insertion in a new page, which cannot be allocated if the database is full. As discussed earlier in Section 2.2.4, this issue can be solved with system transactions [Gra12, GGS16].

The authors of C-ARIES [SK07] suggest a page-oriented undo mechanism that reuses the page-by-page redo scheduler. After redo on a particular page is finished, the corresponding thread starts performing undo of updates performed *on that page only*. It navigates the per-page log chain, invoking the compensating action for updates of loser transactions and skipping CLRs. To make this technique work correctly, a special CLR is generated for undo actions performed during restart recovery rather than normal transaction abort. Furthermore, the system must keep track of all log records that have been undone in all pages. Note that if the compensating action must be performed on another page, the current thread must block waiting for the complete recovery of the other page. How this synchronization is performed, as well as the possibility of deadlocks in this process, is not discussed by the authors. A prototype implementation of the technique is also not discussed.

In general, performing undo at a page granularity is incompatible with the ARIES principle of logical undo with compensating actions. Because undo actions can be applied to a different page than the one affected by the original update, recovery algorithms should not schedule or organize the undo phase by pages in any way. This has been explicitly suggested by Mohan et al. [MHL+92], and the principle was also formalized by Weikum in the multi-level transaction model [Wei91, WV02].

2.5.4 Early availability

A major disadvantage of the basic ARIES restart algorithm presented so far is that post-failure transactions can only access the database after the whole recovery process is complete. Some

approaches were proposed to enable *early availability* of the database, i.e., enabling the execution of post-failure transactions before complete recovery after a system failure.

In the original ARIES paper, two techniques for early availability are proposed [MHL+92]. In the first one, called *deferred restart*, a predefined critical subset of the database pages, e.g., catalog pages or small metadata tables, can be recovered in a first stage of the redo pass; other pages, of so-called *offline objects*, are then kept in a separate data structure for later recovery. In the undo phase, only updates for which the compensating action can be guaranteed to affect the same page can be rolled back; for those that cannot, locks are reacquired in exclusive mode to protect the uncommitted data. Note that this requirement is quite restrictive—most updates (e.g., index operations, record insertion, etc.) require logical undo, and thus there is no such guarantee. After this first stage of recovery, pages of the critical subset which do not contain any update of loser transactions (i.e., uncommitted data) can be accessed by new transactions. Recovery of the remaining offline pages is then performed in the background.

The second ARIES technique for early availability enables new transactions after the redo phase, concurrently with the undo phase. This requires performing the redo pass from the lowest of redo and undo low-water mark and reacquiring all locks of loser transactions—these are known from the log analysis pass. After the redo pass is done, new transactions can be admitted into the system; if they attempt to access uncommitted data that has not been rolled back yet, they will block waiting for the corresponding lock to be released by the undo pass running in the background. Therefore, whenever a lock is granted, the acquiring transaction has the guarantee that the corresponding data object has been correctly recovered.

A further optimization to ARIES restart has been proposed by Mohan to enable new transactions during the redo phase [Moh93]. It relies on the *commit LSN* proposed earlier to optimize two-phase locking [Moh90b]. The commit LSN is the LSN of the oldest update performed by all active transactions; during restart recovery, it includes the loser transactions as well. After log analysis is complete, a new transaction can access a given page p during the redo and undo passes if: (i) p is not in the dirty page table; and (ii) the page LSN of p is lower than the commit LSN. While simple to implement in practice, this technique is not very effective because data that is most likely to be needed by new transactions, i.e., the application's *working set* [Den83], is also most likely to be unavailable, as it probably requires recovery. Furthermore, because the redo pass is a forward log scan, such important data is only released to the application at the very end of restart recovery.

The work of Levy and Silberschatz on *incremental recovery* [Lev91, LS92] was among the first to focus primarily on the problem of early availability during recovery. Unlike the techniques discussed so far, their approach attempts to schedule recovery actions according to the needs of post-failure transactions, so that most important data items are recovered first. Their first approach [Lev91] relied on page-level locking, which simplifies recovery because it allows page-based, physical undo. The technique uses a data structure similar to the dirty page table of ARIES to keep track of pages in need of (both redo and undo) recovery[8]. During restart, transactions accessing a page that is found in this table trigger the page-oriented redo and undo of that page only. To retrieve log records pertaining to a given page, the per-page log chain is used, although the authors also propose a more explicit grouping of log records by page identifier [Lev91].

In a second version of the incremental recovery approach [LS92], Levy and Silberschatz extend the mechanism with a no-steal eviction policy and a log-based propagation strategy. Since a no-steal

[8]The original work proposes maintaining this table in non-volatile memory, but it can also be recomputed during log analysis with indirect checkpoints; thus, non-volatile memory is not strictly required.

policy implies no undo recovery during restart [HR83], page-based redo suffices for correct recovery, and the on-demand mechanism discussed above can be used to provide early availability. However, their revised approach still requires page-level locking because it does not address the issue of recoverability of multi-level actions [Wei91]. In other words, operations that affect multiple pages, such as system transactions, cannot be recovered correctly with a purely page-based approach; thus, logical undo with compensating actions is still required to support record-level locking[9].

To support record-level locking, Levy and Silberschatz propose a multi-level recovery scheme with a higher-level, logical undo log; however, the scheme requires cumbersome bookkeeping of transaction-to-page mappings as "recovery units" and substantially reduces the benefits of incremental recovery [LS92] . Details of this extended algorithm and its correctness, as well as an empirical evaluation of an implemented prototype, are not provided by the authors.

In conclusion, all previous approaches for early availability during restart fall into one of two categories:

(1) Support record-level locking but restrict early access with on-demand recovery to data known to not require any undo recovery.
(2) Support unrestricted on-demand recovery at the page granularity but require page-level locking during normal processing.

The *instant restart* algorithm presented in Chapter 3 improves the state of the art by combining record-level locking with unrestricted on-demand recovery.

2.6 Restart with command logging

So far, the discussion of logging and recovery algorithms has focused on physiological logging, since it is not only adopted by ARIES but also the fundamental assumption of the contribution in this thesis. However, some recent in-memory database system designs propose an extreme form of logical logging called *command logging* [MWMS14], whose main goal is to reduce logging overhead to a bare minimum.

It is a well-known trade-off of logging and recovery algorithms that operations can be logged at higher levels of abstraction (e.g., an SQL statement), which produces more compact, coarse-granular log records [HR83]. However, this comes at the cost of more costly or complex checkpoint and propagation schemes, because the materialized database must be kept in a state of consistency that is compatible with the granularity of logged operations. For instance, if logged operations are SQL statements, than the materialized database must be maintained atomically with respect to SQL statements—e.g., a tuple cannot be inserted in a table if a corresponding record is not inserted in a secondary index, or if a certain post-insertion trigger is not executed completely.

The introduction of command logging in the H-Store database system [MWMS14] was motivated by the observation that logging accounted for ~30% of CPU overhead in a typical OLTP workload [HAMS08]—at least in the open-source prototype system used in the paper. In command logging, each log record describes a single transaction, encoded as a stored-procedure identifier and the arguments used to invoke it. Because this is much more compact than producing one physiological log record for each operation of a transaction (potentially hundreds or thousands), the authors expect that most of the 30% of CPU cycles mentioned above can be saved and spent for transaction execution instead.

[9]The issue of no-steal policies with on-demand recovery and record-level locking will be investigated in more detail in Chapter 4.

However, 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—and maintaining it requires expensive checkpointing procedures that were discussed in Section 2.3.3. Because maintaining transaction-consistent checkpoints is expensive, they must be taken sparingly to not affect performance negatively—otherwise the 30% overhead saved on logging is spent instead on checkpointing. 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.

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 [MWMS14]. If a transaction must roll back in the middle of its execution, the (in-memory) database state is *by definition* in an action-consistent state, which implies that physical or physiological log records are required for undo. Thus, the logging overhead eliminated with command logging is the insertion of these log records in a centralized log buffer and their propagation at commit time. These overheads, however, can also be mitigated in physiological logging approaches, e.g., with well-balanced log bandwidth, scalable log managers [JPS+12], and distributed logging [TZK+13, WJ14]. With these concerns properly addressed, the advantages of command logging seem less compelling.

Finally, command logging is a less flexible approach for applications because, besides the requirement that every transaction must be a stored procedure, it requires deterministic transaction execution [TA10]. Since winner transactions are literally re-executed during recovery, maintaining ACID properties requires that they produce the exact same effects as they did in their original execution. This precludes the use of external or randomized inputs as arguments of a transaction, which is hard to automatically verify in practice.

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.

2.7 Non-volatile memory

Over the past decade, several approaches for storage and transaction management on non-volatile memory (NVM) have been proposed. This section reviews five of these approaches, which are quite different in their design goals and trade-offs, and thus chosen as good representatives for similar approaches in their own equivalence classes. Some of the approaches considered here attempt to leverage the byte-addressability and persistence of NVM devices to minimize traffic to persistent storage (also known as *write amplification*) as well as the memory consumption with reduced redundancy. However, as discussed below, these advantages usually come at the cost of performance (despite some empirical measurements showing improved performance over non-optimized implementations of traditional techniques) and reliability (e.g., by precluding support for media recovery). Other approaches retain the write-ahead logging paradigm while using NVM to optimize runtime performance and simplify certain aspects of recovery.

2.7.1 NVRAM Group Commit

Pelley et al. proposed one of the first studies and new approaches of transaction management and recovery on NVM [PWGB13]. As a baseline, their work proposes a page-based, in-place update strategy with a force policy, in which every modification is immediately persisted while holding an exclusive page latch. To provide atomicity, per-transaction undo logs are also maintained in

NVM and synchronized (i.e., propagated) on every modification. This approach performs page-wise writes to the database but exploits NVM byte-addressability for the undo logs. It is a very simple and effective strategy, but the authors report that it can only perform reasonably well if NVM latency is relatively close to that of DRAM (the authors report 1 μs as target latency [PWGB13], which is fairly reasonable given current NVM latency estimations [OLN+16]). Even if NVM latency is low enough, one disadvantage of this approach is the substantial write amplification caused by page-wise writes on every modification.

As an improvement over the baseline in-place update mechanism, Pelley et al. suggest *NVRAM Group Commit* [PWGB13]. It reduces the overhead of immediately forcing every modification by committing transactions in batches, using transaction-oriented checkpoints [HR83]. Because this implies a force policy, no redo recovery is required, as in the baseline approach. Batches of transactions are persisted atomically using a two-step technique that resembles shadow paging [Lor77] at the cache-line granularity during transaction execution and a differential-file method [SL76] at commit time. While transactions execute, every modification of a cache line is performed on a copy in a volatile *staging buffer*; a bitmap keeps track of modified cache lines. At commit time, i.e., a transaction-oriented checkpoint, the contents of the staging buffer are copied into a persistent differential file, which is then propagated to the materialized; if a system failure occurs, the copy process is simply resumed from the beginning.

This approach has two major performance concerns. First, transaction execution is slowed down due to the maintenance of shadow copies of cache lines and the additional bitmap lookup. Second, transaction-oriented checkpoints require quiescing transactions while the write set of every transaction is copied twice: once from the staging buffer into the differential file, and then from this into the materialized database. Given these substantial overheads, it is unclear why the measurements performed by the authors suggest up to 50% improvement over a traditional ARIES implementation on NVM [PWGB13]. This is justified in the paper by the elimination of the thread contention induced by page cleaner and log flusher threads, which heavily depends on how these services are implemented and scheduled at runtime—see Chapter 4 for a thorough discussion on efficient page cleaning.

Lastly, as in most implementations of the force policy, the lack of redo logging implies lack of support for media recovery.

2.7.2 NV-Logging

NV-Logging [HSQ14] is an approach focusing on the improvement of the logging subsystem by exploiting NVM. Unlike other approaches reviewed here, it assumes that the materialized database is stored on HDD or SSD and cached in DRAM using a traditional buffer pool. The authors claim that using NVM for logging is a more cost-effective solution than using NVM to bypass the DRAM cache or simply using it to persist both the database and the log. This is because in systems with enough main memory to contain the working set of applications and sufficient page cleaning bandwidth, log writes become the main bottleneck of transaction throughput scalability. If this bottleneck is addressed appropriately, a small amount of NVM can bring substantial performance improvements.

The NV-Logging approach reduces log-related bottlenecks in multiple ways. First, while a centralized log buffer is still required to ensure a global LSN order—which is required since the authors use ARIES-based recovery—it only consists of fixed-length entries that point to actual log record contents in NVM. While this does not eliminate a global point of contention for logging operations, it significantly reduces the critical-section overhead in comparison with centralized log buffers [JPS+12]. To support transaction rollback and flush the relevant log records at commit time, each

transaction maintains a linked list of its produced log records with pointers to their allocated memory in NVM.

One potential issue with this approach is that it shifts some of the burden of centralized log buffering into the persistent memory manager, which has to provide persistent pointers to variable-sized log records in a thread-safe manner. This is mitigated in the paper by pre-allocating all log objects in NVM when the buffer of log entries is allocated. However, this leads to substantial memory fragmentation, since log records tend to vary in size from a few bytes all the way to multiple pages. The authors do not discuss the problem of fragmentation, but they mention the use of asynchronous log archiving of NVM logs into SSD or HDD, which helps to maintain a low NVM footprint. The implications of this archiving technique on transaction and system failures are not discussed.

Another approach that optimizes traditional write-ahead logging with NVM was proposed by Wang and Johnson [WJ14]. Unlike NV-Logging, they propose full-fledged distributed logging on NVM among sockets of a many-core CPU. They discuss the interesting issues that arrive with distributed logging in general, such as guaranteeing correct recovery with partially ordered log records, i.e., no global LSN order, how to avoid "holes" in the log, and the trade-offs of partitioning log buffers by pages or by transactions.

In general, using NVM for logging is a very effective solution to improve transaction throughput for memory-resident workloads. Furthermore, it requires little change on existing codebases and does not entail any loss of reliability features, performance, or functionality.

2.7.3 Write-behind logging

Write-behind logging (WBL) [APP16] is an approach based on a force policy for commit processing with multi-version storage, i.e., append-only rather than in-place updates. It leverages the byte addressability of NVM to use records (or, more precisely, records aligned to cache-line boundaries) as the unit of propagation during commit. Since propagation relies on the CPU cache eviction policy, uncommitted updates can reach persistent storage, and an epoch-based[10] logging scheme is used to determine loser transactions during restart. Each group commit invocation determines an epoch interval (c_p, c_d) such that all transactions with epoch lower than c_p were made durable and no transactions with epoch above c_d have made any changes to the database yet. This pair of values is then written into the log, after all changes older than c_p have been persisted—thus the "write-behind" qualifier. Long-running transactions that span multiple such intervals must have their epochs logged individually.

Restart in the WBL approach is very short because it only entails determining the (c_p, c_d) intervals, which corresponds to determining the set of loser transactions—all transactions with epoch in this interval are loser transactions. Each system failure adds a new interval of loser transactions, and thus the current list of intervals is periodically checkpointed and reconstructed in the log analysis phase; an asynchronous garbage collector is required to erase modifications of loser transactions and delete entries from this list, which also enables log truncation. In order to hide their changes from new post-failure transactions, the timestamp of each accessed record is checked; if it falls within one of the loser-transaction intervals, the older version is accessed instead. Therefore, the mechanism only works with multi-versioned (i.e., append-only) storage, where each record points to its previous version.

While the WBL scheme substantially decreases recovery time with its force policy, the overhead on normal processing is increased for multiple reasons. First, while the actual log is indeed very

[10]The original paper uses the term "commit timestamp" to be consistent with the terminology of the concurrency control protocol; here, the term "epoch" is used because it is consistent with similar approaches discussed in this chapter.

compact, the multi-versioned storage is essentially an embedded form of logging—before- and after-images of each record are always available in the materialized database. Thus, the claim that generating such images is faster than generating log records does not always hold, especially when considering that log records are generated in thread-local buffers and appended to the log buffer in bulk [JPS+12], while new record versions must be immediately inserted into a global, multi-threaded data structure. Second, when considering the traffic from volatile to persistent storage, the same issue emerges: forcing new versions of records requires essentially the same write bandwidth as forcing redo log records with after-images. In fact, the latter is faster because it is done sequentially, and this is true even when considering writes from volatile to persistent memory, thanks to CPU cache locality. Finally, as discussed extensively in Chapter 4, the overhead of propagation in a traditional WAL approach is negligible because page cleaning requires very little interference on normal processing. Thus, the only advantage of WBL is the reduced consumption and overall (i.e., log and database) traffic to persistent storage; however, this comes at the cost of reduced redundancy, which precludes support for media recovery.

The measurements performed by the authors of the WBL approach show that it performs better, i.e., has higher average transaction throughput, than a WAL approach [APP16]. However, their WAL implementation also uses multi-version storage, when, in fact, WAL is designed for in-place strategies. Previous work by the same authors [APD15] even demonstrates empirically the higher performance of in-place strategies. Furthermore, details of the commit and logging protocol of their WAL baseline implementation are not provided. As demonstrated by previous research on scalable log managers [JPS+12], efficient logging protocols are essential to high performance in WAL approaches.

Lastly, it is worth pointing out that record-based (as opposed to page-based) logging and propagation schemes usually make index and space management quite cumbersome, because the physical location of individual records must be preserved when undoing or redoing an update. The WAL baseline implemented by the authors of the WBL approach [APP16] does not maintain indexes on persistent storage, requiring full index rebuild during recovery.

2.7.4 FOEDUS

FOEDUS [Kim15] is an NVM transactional engine that relies on a "page duality" mechanism, in which every database page exists in both a read-only *snapshot* state or a modifiable *volatile* state. Rather than keeping track of these states with a separate mapping—as done in shadow paging [Lor77], for instance—FOEDUS relies on tree-based data structures with single-incoming pointers, e.g., Foster B-tree [GKK12], and each data-structure pointer contains the addresses of both snapshot and volatile versions. Snapshot pages reside mainly in NVM but can be cached in DRAM, while volatile pages reside exclusively in DRAM.

Logging in FOEDUS is based on Silo [TZK+13, ZTKL14], which uses distributed redo-only logging and an epoch mechanism to establish durability of transactions. The key advantage of this scheme, other than the better scalability of distributed logging, is that the log only contains modifications of committed transactions. FOEDUS exploits the redo-only log to perform asynchronous propagation of updates into the snapshot versions of each page. To achieve that, it periodically scans and aggregates the log, applying updates to the previous snapshot image of each page. To perform this atomically, updates are replayed in separate copies which are *installed* by swapping page pointers at the end. The downside is that this installation requires quiescing transactions for a short period of time (the author mentions 70 ms in his measurements [Kim15]), which unavoidably causes periodic latency spikes.

This propagation mechanism guarantees that snapshot pages are always in a committed state, but not the most-recent committed state. Therefore, restart recovery entails replaying the log and installing new pointers as described above, while the system is unavailable for new transactions. For pages that were not modified since the last propagation, the volatile image can be dropped—thus providing no-steal eviction—and every access goes directly to the snapshot version. To reduce latency for hot-spot, mostly-read pages, such as root nodes of indexes, snapshot pages are additionally cached in DRAM.

Distributed redo-only logging coupled with log-based propagation is a very useful paradigm to optimize database recovery with simple techniques. Chapters 4 and 5 provide further insights and new proposals based on this idea, without restricting to NVM.

2.7.5 FPTree

The *Fingerprinting Persistent Tree* (FPTree) [OLN+16] is a persistent B-tree data structure whose leaf nodes are maintained in NVM and inner nodes in DRAM. Every modification on the tree—insert, delete, and update of a single key-value pair—is executed atomically, in the sense that the operation is thread safe and also immediately persisted in NVM.

The main goal of the approach is to minimize the number of NVM writes by maintaining leaf nodes unsorted, thereby avoiding the memory writes incurred by shifting in-page slots to maintain sort order. To support this technique, leaf nodes are divided into fixed-size slots, and a bitmap keeps track of occupied slots. In order to avoid linear search over all slots to find a key, an array of one-byte hash values—one for each slot—is stored in the node header. These hash values are called *fingerprints*, thus the name of the approach. To locate a key, this array is first searched with a hash value of the looked-up key; only if a match is found, the corresponding slot is inspected and the keys are compared. This scheme does not natively support variable-sized keys; instead, the authors use pointers to externally allocated memory (i.e., outside B-tree nodes) in their experiments with variable-sized keys [OLN+16]. Therefore, memory management cannot be performed within the data structure, which has been traditionally one of the strong selling points of the B-tree [Com79].

Since only leaf nodes are persistent, recovery from a system failure requires scanning the leaves, which are chained together in a linked list, to rebuild inner nodes. In this process, operations that are interrupted by a system failure, such as allocation of new leaf nodes, are undone to avoid persistent memory leaks. This recovery scheme is not transaction-oriented, since it is only concerned with the atomicity and durability of individual operations on a single data structure.

Despite the advantages of less memory consumption and reduction of write traffic to NVM, persistent data structures such as the FPTree suffer from two major drawbacks. The first is that the requirement for persistent atomic operations without shifting bytes for memory management unavoidably hurts performance. Search is more expensive because it requires $O(n)$ comparisons in a node, despite cache optimizations such as the fingerprint array. Also, scans become more expensive because CPU cache locality is reduced without sort order. Furthermore, modifications are significantly more expensive because cache lines must be forced from the CPU cache into NVM for every single operation. The use of variable-sized keys aggravates these problems further due to out-of-node allocation. Lastly, because hot leaf nodes are not cached in DRAM, the higher latency of NVM is incurred whenever there is a CPU cache miss. Studies comparing the performance of such persistent data structures with state-of-the art indexes [Gra11, MKM12, LKN13, KSHL12] and other persistence mechanisms such as WAL, shadow paging, or multi-version storage are not yet available.

The second major drawback of persistent data structures is that they do not suffice as mechanisms for ACID-compliant transaction processing. To support transactions, these data structures must be coupled with additional redundancy mechanisms such as logging or multi-version storage. However, doing so completely defeats one of the main purposes of such persistent data structures, which is the reduction of persistent memory footprint by eliminating redundancy. If logging is chosen to provide transactional consistency, then there is no reason to perform persistent atomic operations on the data structure itself, and an implementation optimized for volatile memory is a better choice. On the other hand, multi-version storage (as in write-behind logging [APP16]) would incur significantly higher overhead on the FPTree approach, both in terms of lookup performance as well as persistent memory management, which is already more expensive than volatile memory management [OLN+16].

2.8 Media recovery

Media failures usually leave database systems unavailable for several hours until recovery is complete, especially in applications with large devices and high transaction volume. Techniques to recover databases from media failures were initially presented in the seminal work of Gray [Gra78] and later incorporated into the ARIES family of recovery algorithms [MHL+92]. Unlike restart after a system failure, techniques for media recovery have not been researched actively in the past decades, and thus the basic ARIES technique remains as the current state of the art. This section briefly summarizes the restore process in ARIES, highlighting its limitations and justifying the need for new techniques.

2.8.1 ARIES restore

In ARIES, restore after a media failure first loads a backup image and then applies a redo log scan, similar to the redo scan of restart after a system failure. Figure 5 illustrates the process. After loading full and incremental backups into the replacement device, a sequential scan is performed on the log archive and each update is replayed on its corresponding page in the buffer pool. A global *minLSN* value (called "media recovery redo point" by Mohan et al. [MHL+92]) is maintained on backup devices to determine the begin point of the log scan; this is essentially the low-water mark of media recovery.

Because log records are strictly ordered by LSN, pages are read into the buffer pool in random order, as illustrated in the restoration of pages A and B in Figure 5. Furthermore, as the buffer pool fills up, they are also written in random order into the replacement device, except perhaps for some minor degree of clustering. As the log scan progresses, evicted pages might be read multiple times, also randomly. This mechanism is quite inefficient, especially for magnetic drives with high access latencies.

As an example scenario for predicting the time required for media recovery, consider a database storage device of 1 TB. At 150 MB/s, a full backup or a restore operation takes about 2 hours. If 5% of all database pages change over a day, the size of a daily incremental backup is 50 GB, or 6.25 million pages of 8 KB. The restore operation for each incremental backup requires (at 1 ms average access time per page) 6,250 seconds or 1 hour and 45 minutes. Assuming that there are two log records per modified page, that the buffer hit ratio during log replay is 75%, and that the miss penalty is a random I/O operation of 4 ms, then each day of log replay takes ~12,500 seconds or 3.5 hours. Therefore, a media failure occurring 7 days after the last full backup may take almost 16 hours to recover (2h full backup + 6 × 1.75h incremental backups + 3.5h log replay). For around-the-clock online businesses, these prospects are frightening. For some practical scenarios, this example may

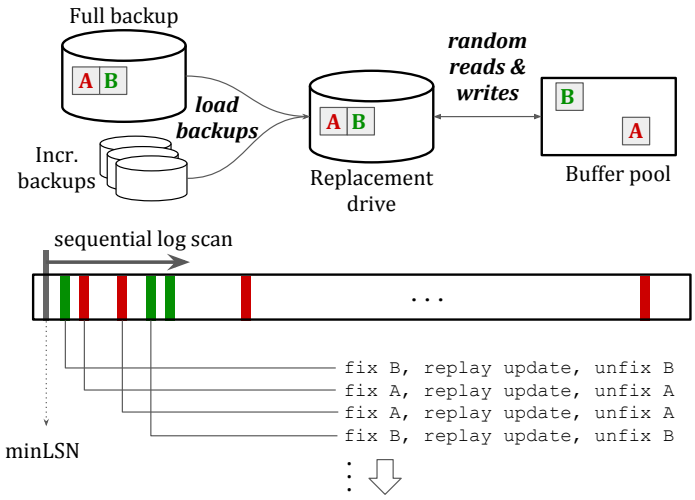

Fig. 5. Random access pattern of ARIES restore

even be considered conservative—if data volumes and transaction throughput are much higher, recovery may easily reach the scale of multiple days.

2.8.2 Incremental backups

The traditional solution to the problem of random I/O during log replay has been to employ incremental backups, also shown in the diagram of Figure 5. While a full backup contains a copy of every page in the database, an incremental backup only copies those pages that have changed since the last full or incremental backup. During restore, incremental backups are loaded after the full backup, overwriting its pages with more recent versions. This way, the length of redo log scan is reduced, and total recovery time is expected to be shorter.

The state-of-the-art algorithm for producing incremental backups was proposed by Mohan and Narang [MN93]. The two key challenges addressed in the algorithm are: (i) to write backups in a fuzzy way, i.e., concurrently with running transactions and without quiescing them; and (ii) to keep track of what pages have not changed since the last backup, i.e., what pages can be skipped by the incremental backup procedure.

To keep track of pages that were modified since the last backup, one bit is reserved in the space-management data structure. A page is only included in the incremental backup if this bit is set to 1, i.e., it has been modified. After a page is copied into a backup, this bit is set to 0; then, it must be set to 1 again as soon as a transaction modifies the page.

In the first stage of the incremental backup procedure, a system transaction collects a list of pages that will be included in the backup. It does so by inspecting the bits described above; if a value 1 is found, the corresponding page identifier is saved in a list and the bit is set to 0. To maintain the consistency of this bitmap in the presence of system failures, this operation is logged. After this step, each page collected in the list is read and copied into the incremental backup. This preliminary step is required to support fuzzy copying while guaranteeing that updates happening during the copy process are not missed by the next incremental backup.

To avoid the overhead of checking the bitmap on every page update, setting it to 1 if necessary, Mohan and Narang suggest a heuristic based on the log-tail LSN at the time the last incremental-backup system transaction completed. This LSN is saved in a global variable and the bitmap is only checked during a page updated if the page LSN before the update is lower than or equal to this saved value. If it is greater, then a previous update must have already checked the bitmap. While the system transaction is running, every update must perform this check.

Incremental backups have two main disadvantages. The first one is that, in practice, the effectiveness of incremental backups depends on multiple factors, most notably workload skew and the amount of main memory available, i.e., the buffer pool hit ratio observed during log replay. For example, replaying a week's worth of log could be faster than loading six daily incremental backups and then replaying one day's worth of log. Furthermore, even with the optimizations described for the algorithm above, incremental backups are still a costly operation that should be performed sparingly to not affect system performance. Therefore, it is hard to assess the effectiveness of incremental backups, plan their frequency, and estimate the effort of media recovery.

Even assuming that incremental backups are an effective solution to reduce log replay time, a second disadvantage remains: the complexity added to normal transaction processing. As described above, a special system transaction, and therefore its recovery logic, must be implemented to permit fuzzy copying. Furthermore, even with the heuristic described above, every page update must include conditional logic to support incremental backups.

2.8.3 Single-page failures

Hardware failures and corruptions that affect a single page of the materialized database are very common in practice, and they can occur due to multiple reasons. Partial writes of a database page, also known as *torn writes*, are one of the main sources of such localized errors [Moh95]. These can happen if a system failure interrupts a non-atomic write of a page, leaving it corrupted.

Further sources of single-page failures are corrupted disk sectors, failed I/O controllers, or wear-out of flash blocks, and these occur frequently in practice, sometimes with much larger-scale consequences [GK12].

While the issue of partial writes can be addressed in practice with specialized hardware such as battery-backed I/O controllers, generally-applicable recovery techniques for the other sources of single-page failures discussed above are not available within the ARIES framework. Instead, Mohan et al. propose invoking the media recovery procedure restricted to the failed page only [MHL+92], and this is also supported by modern database products such as SQL Server [Mic17a]. The downside of this solution is that, even if database backups are indexed, so that a single backup page can be retrieved in constant or logarithmic time, full archive log replay is still required to bring the page to its most recent state.

2.8.4 Early availability

Similar to the techniques for early availability discussed in Section 2.5.4 for restart after a system failure, existing media recovery techniques also lack the support for on-demand, incremental restore of fine-granular data objects. ARIES restore is not incremental and requires full recovery before any data can be accessed—on-demand schedules are not possible and there is no prioritization scheme to make most needed data available earlier. As shown in the example of Figure 5, the last update to page A may be at the very end of the log; thus, page A will be out-of-date until almost the end of the long log scan. Some optimizations may alleviate this situation (e.g., reusing checkpoint information), but there is no general mechanism for incremental restoration. Furthermore, even if pages could

somehow be released incrementally when their last update is replayed, the hottest pages of the application working set are most likely to be released only at the very end of the log scan, and probably not even then, because they might contain updates of uncommitted transactions and thus require subsequent undo.

2.8.5 Replication

Given the extremely high cost of media recovery in existing systems, replication solutions such as disk mirroring or RAID [BG88, CLG+94] are usually employed in practice to increase mean time to failure. However, it is important to emphasize that, from the database system's perspective, a failed disk in a redundant array does not constitute a media failure as long as it can be repaired automatically. Furthermore, even assuming that underlying storage is highly redundant, many failures that occur in practice are correlated, so that individual hardware or firmware defects can escalate into multiple devices, servers, or even a whole data center. Restore techniques aim to improve mean time to repair whenever a failure that cannot be masked by lower levels of the system occurs. Therefore, replication techniques can be seen largely as orthogonal to media restore techniques as implemented in database recovery mechanisms.

A substantial reduction in mean time to repair, especially if done solely with simple software techniques, opens many opportunities to manage the trade-off between operational costs and availability. One option can be to maintain a highly-available infrastructure (with whatever costs it already requires) while availability is increased by deploying software with more efficient recovery. Alternatively, replication costs can be reduced (e.g., downgrading RAID-10 into RAID-5) while maintaining the same availability. Such level of flexibility, with solutions tackling both mean time to failure and mean time to repair, are essential in the pursuit of Gray's availability goal [Gra03b], especially considering the impact of human errors and unpredictable failures that occur in large deployments [Gra86, P+02, BSR+06].

3 INSTANT RECOVERY

Instant recovery is a family of recovery techniques for the different classes of failure discussed in Section 2.1. It improves upon ARIES in multiple ways—some of which are the focus of this chapter—while retaining all its major design cornerstones and advantages. The main motivation is to add incremental and on-demand recovery to the approaches discussed in Chapter 2.

A comprehensive discussion of all instant recovery techniques and its numerous applications is provided by Graefe et al. [GGS16]. This chapter focuses on the two main techniques of instant recovery: instant restart, in Section 3.1, and instant restore, in Section 3.4. A related technique, called single-pass restore, is presented as a special case of instant restore for the purposes of this thesis, even though, in a more general context, it is a standalone design that by itself has numerous advantages and applications. Since a conceptual description of the techniques, their design, and their applications is provided in the referred publication, this chapter concentrates on implementation aspects and empirical evaluations in a prototype system.

Sorting and indexing log records is not only the enabler technique of instant restore but also the single most important foundational technique upon which the work in Chapters 4 and 5 is built. Therefore, aspects of log sorting and indexing are presented in detail in Section 3.3.

3.1 Instant restart

Instant restart aims to provide early availability after a system failure, so that new transactions can be executed immediately after the log analysis phase. However, unlike the approaches discussed in Section 2.5, it does not restrict access to data known to not need any recovery; in fact, it prioritizes recovery actions so that data needed most immediately is also recovered first.

The instant restart algorithm uses the same building blocks and actually executes the exact same actions as the ARIES algorithm during recovery. In the redo phase, the same log records are replayed on the same set of pages as in ARIES restart; in the undo phase, the same transactions are rolled back, producing the same logical compensation actions. The key difference is that these actions can be performed concurrently with post-failure transactions, and their access pattern is actually used to guide recovery in an on-demand schedule.

This section briefly discusses the instant restart mechanism and, most importantly, its implementation and evaluation in a prototype system in two parts: Section 3.1.1 discusses normal processing and checkpoints, while Sections 3.1.2–3.1.4 describe the restart algorithm in its three phases—log analysis, redo, and undo. Then, Section 3.1.5 presents a qualitative analysis of the performance instant restart, especially the factors that determine its efficiency in comparison with ARIES. Finally, Section 3.2 presents a quantitative analysis with experiments on the prototype implementation.

3.1.1 Normal processing and checkpoints

In instant restart, normal transaction processing is very similar to ARIES, except for a few minor modifications; these are briefly described below.

First, the per-page log chain described in Section 2.2 is required for on-demand redo. This means that when a page is updated in the buffer pool, the generated log record must contain the previous page LSN value in its `page_prev` field. During restart, as described below, the per-page log chain is used to recover individual pages incrementally and on demand. System transactions that update multiple pages [Gra12] must perform this logic for each page updated. As discussed in Section 2.2, a reasonable design trade-off is to restrict the number of pages updated in a single log record to two, providing additional fields `page_id2` and `page_prev2` to maintain a second log chain. The

roles of each page must be fixed according to the system transaction type; for example, a log record describing a page split could always use the first field for the overflowing page and the second field for the new page.

Checkpoints for instant restart differ from ARIES checkpoints in two ways. First, the dirty page table contains the LSN of the last update on each page, rather than its recovery LSN field. This is another requirement for on-demand redo, as described below. Note that the recovery LSN must still be maintained for each frame in the buffer pool—as described in detail in Chapter 4—because the lowest recovery LSN (i.e., the redo low-water mark) is still used to guide log space recycling.

The second distinction to ARIES checkpoints is that a list of exclusive locks currently acquired by each active transaction must be collected in addition to the dirty page and active transaction tables. These locks enable on-demand rollback of loser transactions while protecting their uncommitted updates from new concurrent transactions. This list is acquired by scanning the lock manager data structures, which can be done in a fuzzy way, i.e., without quiescing transactions.

3.1.2 Log analysis

Instant restart performs offline log analysis just like in ARIES, with the distinction that it reacquires the exclusive locks of loser transactions. Such locks are collected from the latest complete checkpoint and all log records between its begin log record and the log tail. Log analysis can be simplified and lock reacquisition optimized if log analysis scans the log backwards from the log tail to the latest complete checkpoint. This also has the advantage that the LSN of the latest checkpoint does not have to be stored separately—it is always found by backward log analysis.

During log analysis, page-flush log records can be used to remove entries from the dirty page table, thus reducing the number of false positives as discussed in Section 2.5.1. If the page cleaner does not latch buffer pool frames while flushing the page's contents and producing a page-flush log record, i.e., if page cleaning is *asynchronous*, log analysis must compare LSNs before removing an entry from the dirty page table. This is required to eliminate false negatives—i.e., pages that require recovery but are not included in the dirty page table—which result in incorrect recovery.

Figure 6 illustrates a situation that can result in a false negative. Page A is copied with version A_1 into the page cleaning buffer with LSN 100, after which its shared latch can be released. Then, a transaction modifies page A, producing a new version A_2 with LSN 120, and only after that the page cleaner completes the write of the old page image and generates a log record. If log analysis simply assumes that page A is clean because the page-flush log record comes after the latest update on that page, the log record with LSN 120 will not be replayed during redo recovery, and version A_2 will be lost.

To eliminate the false-negative problem while allowing asynchronous cleaning, the dirty page table used during log analysis must contain two LSN values for each entry: *page LSN* and *clean LSN*. The first value is simply the latest update found for each page; in a backward log scan, it is set only once, namely when an entry is first created for a page. The second value is an upper bound for the LSN range known to have been persisted by the page cleaner. Before the page cleaner makes copies of a set of pages to be flushed, it marks the LSN of the current log tail; then, it flushes those pages and produces a page-flush log record containing the IDs of the flushed pages and the log-tail LSN marked earlier—this is the clean LSN.

When backward log analysis finds a page-flush log record, it simply updates the clean LSN value of each flushed page. Note that the clean LSN is not the LSN of the page-flush log record itself, but an LSN value stored inside it. When log analysis finishes, it can finally remove all entries in which the clean LSN is higher than the page LSN, thus guaranteeing that no false negatives are produced.

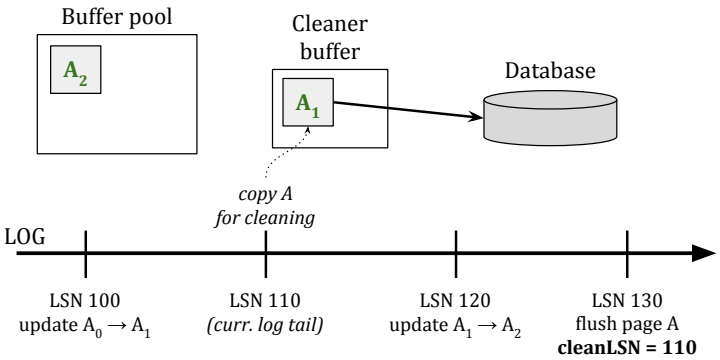

Fig. 6. Asynchronous cleaning with *clean LSN* to avoid false negatives

In Figure 6, the clean LSN of page A is 110; since it is lower than its page LSN 120, A will be kept in the dirty page table.

Asynchronous cleaning also requires additional LSN fields and special logic to maintain correct recovery LSN values in the buffer pool (required to compute the redo low-water mark for log space recycling); this issue is discussed in detail in Chapter 4. In essence, asynchronous cleaning precludes the use of simple Boolean flags to determine the propagation state of a page, for both log analysis and checkpoints.

Algorithm 1 shows the pseudo-code for the log analysis algorithm with a backward log scan, using the techniques presented here. The auxiliary procedure ProcessPageFlush is shown in Algorithm 2. The clean-LSN mechanism used to avoid false-negative dirty pages can also be adapted to a forward log scan without loss of generality.

3.1.3 Redo recovery

Redo recovery in instant restart differs from the redo phase of ARIES restart in multiple ways. First, redo actions are naturally executed in parallel and concurrently with post-failure transactions, without any restrictions to which data objects can be accessed. Second, it enables page-by-page redo by replaying log records of a single page using the per-page log chain, in the same way as repair following a single-page failure [GK12]. Third, the order in which pages are recovered is determined primarily by the demands of post-failure transactions. Fourth, instant restart redo is actually a generalization of the redo phase in ARIES restart, because it permits both on-demand, page-oriented redo as well as a log scan starting from the redo low-water mark.

Figure 7 illustrates log replay restricted to single pages using the per-page log chain. The top diagram (a) shows the process of single-page repair, in which the LSN of the most recent update, i.e., the expected page LSN after recovery, is retrieved from a page recovery index [GK12]. With self-healing indexes [GKS12a], this expected LSN information can be stored in the parent-to-child pointer of a B-tree as well, but the principle is the same. Not shown in the diagram is the backup device, from which an older image of the page is retrieved, the page LSN of which determining where the log chain traversal stops. To control log space recycling, the backup low-water mark is stored alongside each backup; this also establishes a global limit for how far backwards the log chain traversal can reach.

On the bottom diagram (b), the same page-oriented log replay procedure is used in instant restart. The difference here is that the expected page LSN is retrieved from the dirty page table computed

Algorithm 1 Backward log analysis algorithm

```
1:  procedure BackwardLogAnalysis(logTailLSN)
2:      dirtyPages ← ∅, activeTxns ← ∅
3:      seenChkptEnd ← false
4:      scan ← BackwardLogScanner(logTailLSN)
5:      while scan.hasNext() do
6:          lr ← scan.nextLogRecord()
7:          if lr.type = chkpt_begin and seenChkptEnd then
8:              load checkpoint data and merge it with dirtyPages and activeTxns
9:              break
10:         else if lr.type = chkpt_end then
11:             seenChkptEnd ← true
12:         else if lr.type = page_flush then
13:             ProcessPageFlush(lr, dirtyPages)
14:         else if lr.type = txn_end then
15:             UpdateActiveTxns(lr, false, activeTxns)
16:         else if lr.isTxnUpdate() then
17:             UpdateDirtyPages(lr, dirtyPages)
18:             UpdateActiveTxns(lr, true, activeTxns)
19:         end if
20:     end while
21:     delete entries e from dirtyPages such that e.clean_lsn > e.page_lsn
22:     delete entries e from activeTxns such that e.active = false
23:     return dirtyPages, activeTxns
24: end procedure

25: procedure UpdateDirtyPages(lr, dirtyPages)
26:     p ← dirtyPages[lr.page_id]
27:     if p = null then
28:         p ← {page_lsn : LSN_NULL, clean_lsn : LSN_NULL}
29:         dirtyPages.insert(lr.page_id, p)
30:     end if
31:     if lr.lsn > p.page_lsn then
32:         p.page_lsn ← lr.lsn
33:     end if
34: end procedure

35: procedure UpdateActiveTxns(lr, isActive, activeTxns)
36:     t ← activeTxns[lr.txn_id]
37:     if t = null then
38:         t ← {active : isActive, last_lsn : lr.lsn}
39:         activeTxns.insert(lr.txn_id, t)
40:     end if
41:     if t.active = true then
42:         acquire X-lock on the identifier extracted from lr on behalf of lr.txn_id
43:     end if
44: end procedure
```

Algorithm 2 Processing page-flush log records during log analysis

```
1:  procedure ProcessPageFlush(lr, dirtyPages)
2:      pageIDs ← deserialize list of page IDs from lr
3:      clean_lsn ← deserialize clean LSN from lr
4:      for all pid in pageIDs do
5:          p ← dirtyPages[pid]
6:          if p = null then
7:              p ← {page_lsn : LSN_NULL, clean_lsn : LSN_NULL}
8:              dirtyPages.insert(pid, p)
9:          end if
10:         if clean_lsn > p.clean_lsn then
11:             p.clean_lsn ← clean_lsn
12:         end if
13:     end for
14: end procedure
```

during log analysis. Since recovery is only required for dirty pages, this table is substantially smaller than the page recovery index, and is thus kept in main memory. Lastly, the redo low-water mark establishes a global limit for log chain traversal of all pages, guides log space recycling, and enables prefetching of the log portion that will be used for redo.

Redo recovery of a given page is triggered by a fix operation in the buffer pool. This requires a way to determine, upon a page miss, if that page might require recovery or not, i.e., if it was flagged as a dirty page during log analysis. One way to achieve this is to maintain the dirty page table as an auxiliary data structure either in the buffer pool or in the I/O manager, as illustrated in the bottom part of Figure 7. In this case, this data structure must be merged with the dirty page table when taking a checkpoint. An alternative implementation reuses the frame control blocks of the buffer pool, by initializing, for each dirty page, an empty frame with the expected page LSN and a Boolean flag indicating the need for single-page recovery when that frame is first fixed. This implementation has the advantage of eliminating the additional data structure lookup of the first approach and requiring no changes on checkpoint code.

The performance of single-page log replay largely depends on the length of the per-page log chain of each page being recovered. The recovery of frequently updated hot-spot pages may require hundreds of random reads on the log device, which, in turn, increases the latency of the buffer pool fix operation and thus of currently active transactions in general. The most obvious solution to this problem is to clean such pages frequently, as discussed in depth in Chapter 4. However, such repeatedly writing a hot-spot page might not be appropriate for the I/O policies established by a system administrator, as it increases write amplification and causes additional stress on the I/O subsystem.

Instant restart presents an alternative solution: to let the dirty page linger in the buffer pool but restrict the length of the per-page log chain with *page-image log records*. These contain compressed images of an entire page's contents, and thus they impose a termination point for single-page log replay. These log records, which can be generated by system transactions, do not perform any update on the page, but the page LSN value in the buffer pool must be adjusted accordingly to maintain the per-page log chain correctly. Alternatively, they can be generated by user transactions

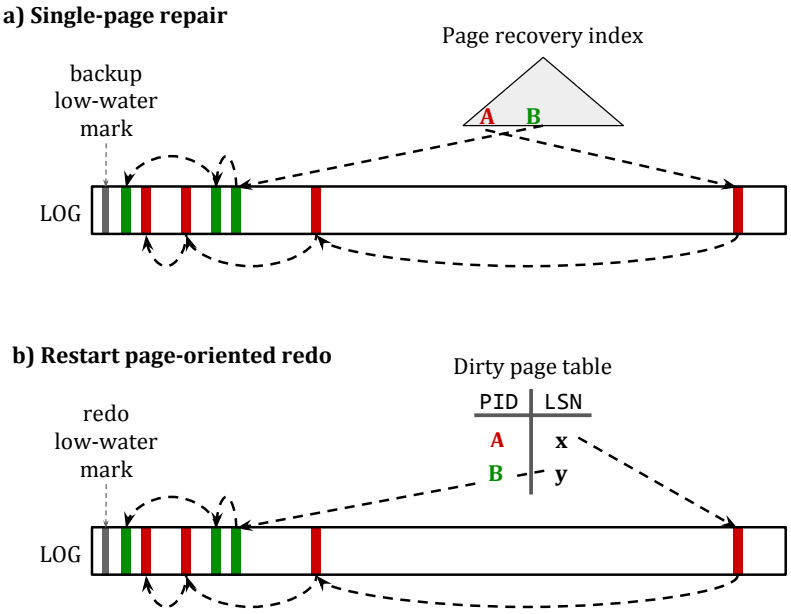

Fig. 7. Log replay in (a) single-page repair and (b) instant restart

as whole-page after-images of a particular update, replacing the log record that would be generated for that single update.

The generation of page-image log records can follow a variety of policies depending on time, number of log records generated, log volume in bytes, etc. One simple approach, which was implemented in the prototype system of this work, is to keep track of the number of log records generated for each page in the buffer pool. This can be a counter in the frame control block, which is updated on every update together with the page LSN. Whenever it reaches a certain threshold, 2× the page size), a page-image log record is generated and the counter is reset.

Figure 8 illustrates the use of page-image log records and how they restrict log chain traversals during single-page log replay. Also note that, in this example, the redo low-water mark has also been increased thanks to the new log records.

Note that the applicability of page-image log records goes beyond optimizing single-page log replay. Since they enable the restoration of a whole page, fetching an older version of a dirty page from the database is not strictly required, or it can be done while traversing the per-page log chain. Furthermore, page-image log records contribute to the increment of the redo low-water mark, which guides log space recycling. These techniques present interesting opportunities for page cleaning and propagation policies, which will be exploited with novel techniques in Chapter 4.

One obvious drawback of page-image log records is that they increase log size; thus, its applicability and effectiveness are subject to a system-wide trade-off between recovery efficiency, log space recycling, I/O load balancing, and page cleaner policies.

Like in ARIES recovery, instant restart redo follows the repeating-history paradigm [MHL+92]. Thus, it simply replays all (physiological) log records of a page up to the most recent one, and then it immediately makes the page available for fix operations in the buffer pool. The possible need for undo recovery is a separate concern, addressed by logical, transaction-oriented undo,

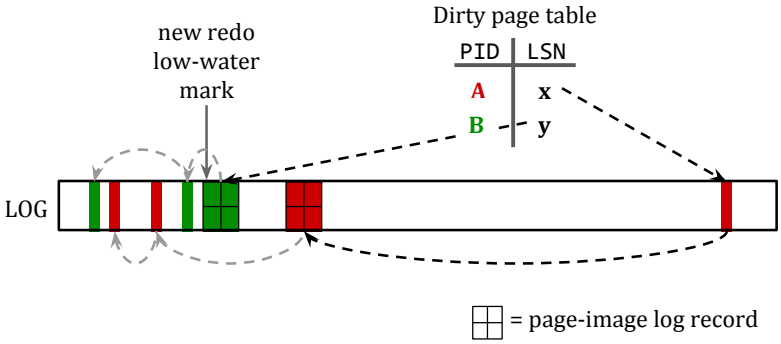

Fig. 8. Restricted per-page log chain traversal with page-image log records

which is discussed below. This paradigm is also formalized in the multi-level transaction model [Wei91]. Given this separation of concerns, the single-page recovery process is triggered by any fix operation in the buffer pool, regardless of its origin—new post-failure transactions, rollback of loser transactions, page cleaning, backups, etc. This opens the possibility for an additional form of asynchronous bulk recovery, other than the traditional redo log scan of ARIES: a background thread that simply fixes all dirty pages computed during log analysis.

3.1.4 Undo recovery

Undo recovery in instant restart follows the same on-demand pattern of redo recovery, but the unit of recovery is a transaction rather than a page and the triggering action is a lock conflict rather than a fix operation in the buffer pool. As shown in Algorithm 1, locks of loser transactions are reacquired before log analysis finishes. These locks serve two main purposes. First, they guarantee that changes made by loser transactions are isolated from post-failure transactions. Lock reacquisition for this purpose is also present in the ARIES optimizations for early availability [MHL+92, Moh93], with the difference that locks are reacquired in the redo phase rather than log analysis.

The key improvement of instant restart undo over ARIES is that the reacquired locks serve a second purpose: they trigger the rollback of loser transactions whenever a post-failure transaction attempts to acquire them. Such an attempt results in a special kind of lock conflict, in which the rollback of the loser transaction currently holding the lock is triggered. This requires an additional Boolean flag on lock manager entries to determine whether an acquired lock belongs to a loser transaction or not. As in ARIES restart, rollback makes use of the last LSN of each loser transaction computed during log analysis and the per-transaction log chain guides the process. When the loser transaction is rolled back, the lock is automatically granted to the new transaction by the lock manager.

Similar to the dirty page table discussed for redo recovery, the active transaction table can either be maintained on a separate data structure or reuse the transaction manager with special entries for loser transactions. This latter approach potentially eliminates need for the loser flag in the lock manager, since rollback-triggering lock conflicts can be determined in the transaction manager as well.

As loser transactions roll back, they fix pages in the buffer pool, which might trigger redo recovery as described earlier. Thus, redo and undo recovery are kept decoupled and automatically coordinated with buffer pool fixes and lock requests.

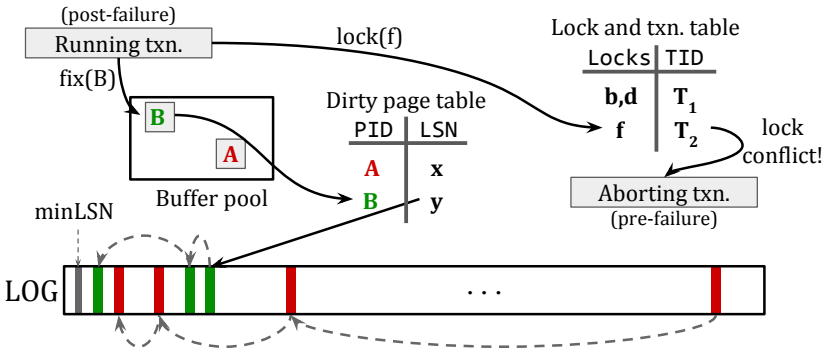

Fig. 9. Instant restart with page-oriented redo and transaction-oriented undo

Figure 9 illustrates the processes of on-demand redo and undo recovery. In this example, log analysis has produced a dirty page table with pages A and B, with expected page LSNs x and y, respectively. Furthermore, it has produced an active transaction table with transactions T_1, holding locks on keys b and d, and T_2, holding a lock on the key f. A post-failure transaction, shown on the top-left corner, fixes page B, which triggers single-page log replay following the per-page log chain. This transaction also attempts to lock the key f, which results in a lock conflict with the loser transaction T_2. This triggers the rollback of T_2, which executes the required compensation actions like any other transaction. While not shown in the diagram, rollback is guided by the per-transaction log chain, just like in ARIES. If T_2 attempts to fix page A, it will also trigger redo on it, just like performed for B in the diagram.

Further details of the instant restart algorithm are provided in its original publication [GGS16]. As mentioned earlier, the on-demand redo and undo mechanisms presented here can be combined with asynchronous recovery threads that perform either log-based or page-based redo. Coordination among these concurrent recovery actions is transparent to each invoking thread, as it relies on the fix operation of the buffer pool as well as the lock manager. Such coordination also supports the other classes of failure discussed in Section 2.1.

3.1.5 Performance expectations

Before presenting the empirical evaluation of instant restart in Section 3.2, it is worth exploring the factors that determine the efficiency of instant restart in comparison with ARIES restart in a qualitative analysis. The following paragraphs provide a brief attempt of such an analysis. A thorough analysis of all factors involved, possibly enhanced with an analytical model, would constitute extensive work and is thus out of the scope of this thesis.

If transaction throughput is plotted as a function of time after a system failure, it has the shape of a sigmoid curve known as the *logistic function*, shown in Figure 10. This function describes the behavior of certain natural processes that have exponential growth in an initial phase, but then transition into logarithmic growth until stabilizing at a maximum saturation point. It was first studied as a model for population growth [Wik17].

Figure 10 shows an example of the logistic function as applied in restart recovery. It highlights three stages of the recovery process: offline phase, warm-up, and steady state. In the offline phase, the system cannot process any transactions, and thus the y-axis value is 0. In the warm-up stage, transactions start executing and their working set is fetched into the buffer pool, which increases

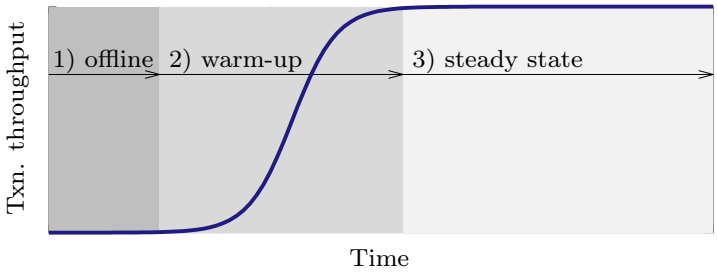

Fig. 10. Logistic function pattern and phases of restart

throughput at the rate given by the logistic function. Finally, steady state, i.e., maximum throughput, is reached when either the buffer pool is full or the working set has been entirely cached in it. Database restart always has these three distinctive stages, even if no recovery work is required (i.e., clean restart). Thus, restart efficiency can be analyzed in terms of the duration of the first two stages.

In traditional ARIES restart (without the early availability optimizations discussed in Section 2.5.4), all recovery work is performed in the offline stage. In optimized versions of ARIES as well as in instant restart, recovery work is performed in the other two stages as well; the difference is that instant restart performs the vast majority of recovery work (i.e., redo and undo phases) in the second and third stages.

How well instant restart performs in comparison with ARIES depends on a number of factors, which are analyzed below. These factors determine the shape of restart logistic curves, which is a useful abstraction to evaluate recovery performance in a qualitative way.

First, the analysis can be simplified by focusing only on the redo phase. As evaluated in detail in previous work [SGH14], workload characteristics determine the amount of undo work, but typical OLTP workloads such as TPC-C have short transactions; thus, as measured in the experiment of Figure 11, undo costs are usually negligible (and redo costs correlate strongly with the number of dirty pages). Furthermore, as discussed in Section 2.3, proactive system activities such as checkpoints cannot reduce undo costs. Therefore, this analysis focuses on redo costs, but most factors considered apply to undo costs without loss of generality.

Figure 12 shows the general shape of the logistic curves of instant restart and ARIES, as usually observed in practice. In instant restart, the length of the offline phase, i.e., log analysis, is determined solely by the frequency of checkpoints. As a general rule of thumb, if checkpoints are taken every t seconds, than log analysis should last less than t seconds.

The area of regions 1 and 2 highlighted in Figure 12 serve as a measure of recovery efficiency: their area corresponds to the number of transactions that the system "misses" due to a system failure. Thus, the smaller this area, the more efficient restart recovery is. The number of missed transactions in instant restart is given by region 1, and that of ARIES restart is the sum of regions 1 and 2. This assumes that instant restart is more efficient than ARIES—unusual scenarios where that might not be the case are discussed later. The diagram also shows the times in which each restart curve transitions from the offline into the warm-up phase—which occurs much earlier in instant restart.

The steepness of the curve, i.e., how fast maximum throughput is re-established, depends on the following factors: (I) the number of dirty pages computed by log analysis; (II) the distance between the redo low-water mark and the log tail, referred to here as *redo length*; (III) the read latency of the

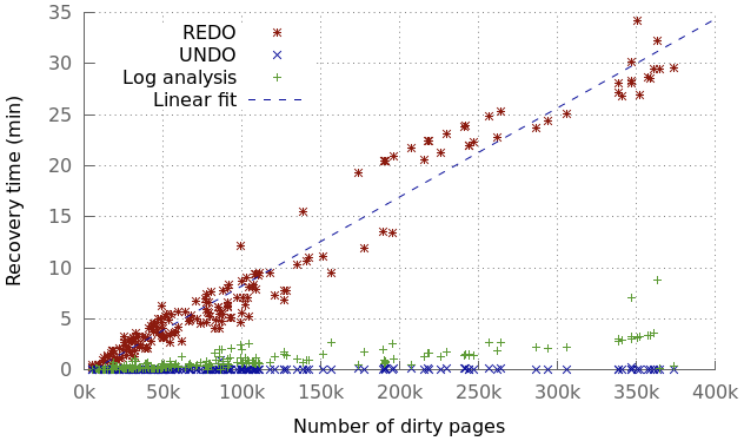

Fig. 11. Duration of log analysis, redo, and undo phases of various restart time measurements plotted as data points (details in [SGH14]). Across all measurements, undo duration is negligible.

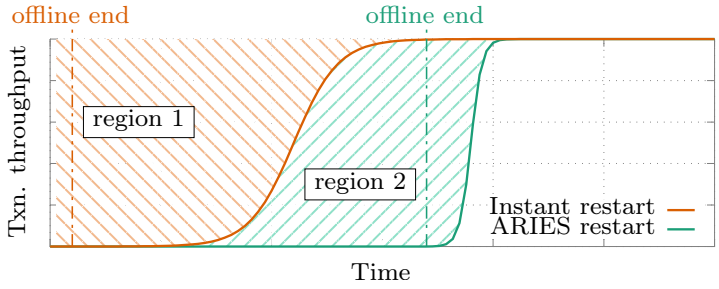

Fig. 12. Logistic function patterns of instant restart and ARIES restart

database device; (IV) the locality of pages in the working set; (V) the read latency of the log device; and (VI) the locality of log records pertaining to the working set.

The first two factors—dirty page count and redo length—depend on the efficiency of the page cleaner, which is discussed in detail in Chapter 4. An efficient cleaner algorithm provided with enough write bandwidth is able to maintain these very low, in which case instant restart and ARIES have very similar, steep warm-up curves. Factors III and IV determine the time it takes to fetch all pages in the working set; this cost is incurred by all restart procedures—even from a clean state—and in ARIES it also exclusively determines the steepness of the curve in the warm-up stage. Because ARIES fetches all the dirty pages in its offline phase, while performing redo, its warm-up phase entails fetching clean pages only. In instant restart, it also determines the steepness of the curve, but not exclusively, as discussed next.

When it comes to comparing the warm-up efficiency of ARIES and instant restart, the key factors to be considered are V and VI—i.e., the log access pattern. This is because the log is accessed randomly during single-page log replay for instant restart, while ARIES scans and replays it sequentially. The efficiency of log replay depends on the size of the I/O unit chosen for a particular log device and on the distribution of log records of dirty pages, i.e., log records that will actually be used for log replay during redo, across the redo length. These two factors are illustrated in Figure 13, which

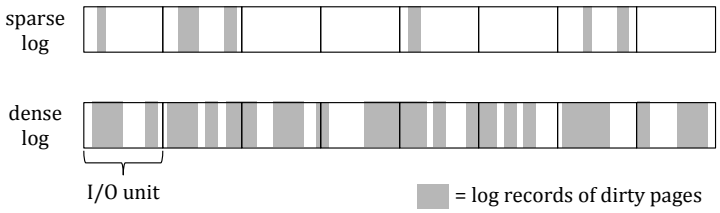

Fig. 13. Possible distributions of dirty-page log records

shows two log fragments divided into fixed-size I/O units; the top one is a *sparse* log, in which log records of dirty pages occur with low frequency and are spatially spread out, and the bottom one is, conversely, a *dense* log.

A sparse log benefits instant restart, as it requires less reads in total—in the diagram of Figure 13, only 4 of the 8 I/O units must be fetched to perform all redo work. If the log device has low read latency, smaller I/O units can be used, making instant restart even more efficient. In ARIES log replay, the effort of scanning a sparse log is the same as a dense log; thus, as long as I/O units are sufficiently large and enough main memory is available to cache the log, instant restart is expected to achieve steady state earlier than ARIES. A sparse log should not occur frequently in practice, because it is an indicator for unusual page cleaning behavior. It can only result from a cleaning policy that deliberately gives low priority to pages that are updated sporadically and also to reducing the redo length. Nevertheless, if log space is not a concern, such a policy might benefit instant restart—Chapter 4 discusses this issue further.

In a dense log, which is more common in practice, the total number of reads should be approximately the same in ARIES and instant restart. Therefore, ARIES should have a steeper warm-up curve than instant restart, given its sequential access pattern. This behavior is shown in Figure 12.

The six factors discussed above (I–VI) contribute to the duration of the warm-up phase, i.e., the steepness of the logistic function. As discussed, ARIES log replay tends to have the edge over instant restart here thanks to the sequential log replay pattern. However, the whole point of instant restart is that, as shown in Figure 12, the warm-up phase *begins much earlier*, so that steady state is reached much earlier, even with a less steep curve. In other words, instant restart offers a trade-off between performing all recovery work offline in a more efficient way versus performing it online but less efficiently. As the experiments later on show, the latter approach usually reaches steady state earlier. Furthermore, even in the unusual cases where instant restart reaches steady state later, it will always be better in terms of *perceived availability*, i.e., the time during which a system is unable to execute *any* transaction after a failure.

One final aspect to consider when analyzing the logistic curves of ARIES and instant restart is skew. As discussed, both approaches incur the same I/O cost in terms of database pages, i.e., the cost to read all dirty pages, mostly randomly. Thus, if access to database pages is skewed, there should be fewer dirty pages, which benefits both approaches equally. In terms of log access, such kind of skew does not affect the density of dirty-page log records, and thus log replay is mostly insensitive to data access skew for both approaches. However, if data access is skewed and the working set *shifts over time*, instant restart is able to exploit this fact and warm up substantially faster.

To illustrate this scenario, Figure 14 extends the dense log of Figure 13 with an additional dimension: the "heat" of log records. In a workload whose working set shifts over time, pages whose log records were produced most recently, i.e., hot pages, are most likely to be accessed by post-failure

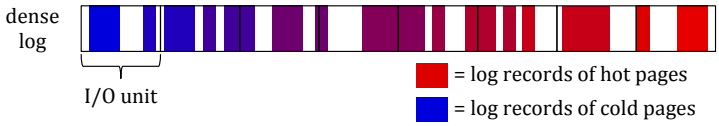

Fig. 14. Heat distribution of page identifiers in the log for workloads with shifting working set

transactions—these are shown in red in Figure 14. Pages accessed least recently (i.e., cold pages, shown in blue), on the other hand, are less likely to be accessed. Instant restart exploits this situation very effectively because the hotter a page is, the earlier it will be recovered and made available to new transactions.

Note that, in practice, heat is not distributed in such a uniform gradient as shown in Figure 14, which is used for illustration purposes—some hot pages may have log records spread across the entire redo length, not just clustered in the most recent portion. Nevertheless, instant restart is still able to recover those pages much earlier. Furthermore, the page-image optimization discussed in Section 3.1.3 makes the log access pattern of instant restart much more sparse and more clustered by heat as shown in the diagram, providing a significant improvement to warm-up speed.

3.2 Restart experiments

This section presents a quantitative analysis of the efficiency of instant restart, in terms of the logistic function as discussed earlier. The basic ARIES restart algorithm (without early-availability optimizations) is used as baseline, which was implemented on the same prototype and under the same workload.

Before presenting the results, the software and hardware environment, as well as the workload used in the benchmarks, is described below.

3.2.1 Environment and workload

The software prototype used in the experiments is the open-source transactional storage manager *Zero*[11], a fork of Shore-MT [JPH+09] with extensive modifications and whole components replaced. The workload used throughout all experiments in this thesis is the TPC-C benchmark as implemented in the *Kits* framework of Shore-MT[12], but adapted to use Foster B-trees [GKK12] as the storage structure for all table and index data. The experiments in this section were performed on a Linux server with the configuration shown in Table 2.

In the following experiments, an initial database of 100 GB (TPC-C scaling factor 750 in this case) is loaded and persisted entirely in the materialized database, so that the log is fully truncated. Then, the benchmark runs with page cleaning disabled, so that the loaded database remains intact, until a certain log volume in gigabytes is reached, after which the system immediately shuts down. This process is repeated to generate different snapshots of the workload with varying log sizes, from 1 to 128 GB in exponential increments. Then, the benchmark is restarted, first using ARIES restart and then instant restart; the number of transactions committed in every second is captured, until the system reaches maximum, steady-state throughput. Because the materialized database is left

[11] http://github.com/caetanosauer/zero
[12] http://sites.google.com/site/shoremt/

Processor	4 × Intel Xeon E7-4830 (48 cores total)
Main memory	1 TB DDR4 2400 MHz
Operating system	Ubuntu Linux 16.04 Kernel 4.4.0
Compiler	gcc 5.4 with -O3 optimization
SSD devices	Samsung 840 Pro 256 GB
HDD devices	Western Digital Caviar Green 2 TB

Table 2. Server used for restart experiments

intact, the size these logs corresponds to the redo length during recovery—this is the first dimension considered in the empirical analysis.

The goal of these experiments is to isolate recovery performance, excluding the effect of the page cleaner. The page cleaner has significant impact on the amount of redo recovery work required during restart, and, because its performance tends to vary significantly during a benchmark, it is best left out of an analysis of recovery efficiency. To compensate for this, the experiments analyze recovery efficiency with varying, pre-determined redo lengths as described above. The performance of page cleaning strategies and their impact on recovery will be evaluated in depth in Chapter 4.

In addition to the redo length, a second dimension considered in the experiments is workload skew. As discussed earlier, skew can make instant restart perform much better than ARIES, because the higher the skew, the greater the impact of on-demand recovery is, as less data must be recovered to achieve near-maximum throughput. ARIES, on the other hand, should be less sensitive to skew—its offline phase lasts approximately the same time, despite warm-up being faster.

As a third dimension, the experiments also use different types of storage device for log and database. Rather than testing all possible combinations, three interesting scenarios are picked: first, with both log and database on flash-based SSDs, which should be one of the most common choices with current technology. Then, the log is stored in DRAM to evaluate how incoming non-volatile memory devices can impact recovery performance. Even if these do not deliver the same latency as DRAM, this experiment is still useful as it extrapolates what can be achieved with very low latency. Finally, the database is placed on a magnetic disk and the log in DRAM; this anticipates a scenario where high-density, high-bandwidth magnetic disks in combination with low-latency non-volatile memory actually replace SSDs entirely.

All restart experiments performed here use a buffer pool large enough to fit the entire working set. As discussed earlier with the logistic function analysis, restart recovery performance is best measured in terms of duration of the offline and warm-up phases. Thus, a very large buffer pool is best suited as a stress test for recovery performance, because it maximizes the amount of recovery work as well as the demand imposed on the system to reach maximum transaction throughput after a failure. Varying the size of the buffer pool would introduce yet another dimension to the analysis without much insight gained, as the general shape of the logistic curves would remain the same, but at different scales.

Lastly, in order to maximize transaction throughput, the workload is tweaked in two ways. First, logical contention is eliminated by assigning worker threads to static partitions of the TPC-C database (i.e., warehouses) and no cross-partition transactions are executed. Second, the system runs in a non-durable mode, in which a commit is acknowledged before the corresponding commit log record is persisted. These two measures provide a substantial improvement in steady-state transaction throughput, thus increasing pressure on the logging and recovery subsystems. As a side

note, the performance of durable mode can match that of non-durable mode if effective logging and group commit strategies are implemented—a concern which is orthogonal to this thesis and has been addressed extensively in related work [HSL+87, JPS+12, WJ14, ZTKL14, HSQ14].

3.2.2 Log and database on SSD

The first experiment, whose results are shown in Figure 15, analyzes restart efficiency with log and database stored on independent SSD devices. A total of eight restarts are shown, with redo length varying exponentially from 1 to 128 GB. The workload has a uniform access pattern, so that each warehouse of the TPC-C benchmark is chosen with equal likelihood—the experiment is thus non-skewed. Each of the eight plots shows the logistic curve for ARIES and instant restart, with the x-axis showing elapsed time in minutes since the failure and the y-axis showing transaction throughput in thousands per second (ktps). The two bar charts on the bottom show the number of missed transactions in each restart procedure (defined earlier in Figure 12 as regions 1 and 2) and their difference, which can also be visualized by the shaded region in the logistic function plots.

The results show that, as expected, instant restart starts processing transactions much earlier—from just under a minute and a half in the first restart up to 10 minutes in the last one. The warm-up curve looks almost exactly the same for ARIES in every experiment (note for example the small dimple at the two-minute mark in the first experiment, which occurs consistently in the other experiments). This is expected because warm-up does not entail any recovery work—the database is fully recovered after the offline phase—and thus it does not depend on the redo length. Instant restart, on the other hand, has a less steep warm-up curve as the redo length increases, which is also expected because warm-up and recovery work are intertwined.

As the bottom plots show, the number of missed transactions grows faster with redo length in ARIES restart, so that the difference to instant restart also increases. In the 128-GB experiment, more than eight million transactions that could not be processed by ARIES restart were processed by instant restart. This difference corresponds to four and a half minutes of service at maximum, steady-state throughput, with two orders of magnitude of improvement—i.e., two "nines"—on perceived availability (i.e., time to first transaction).

To visualize the improvement of two hours of magnitude on perceived availability, Figure 16 shows a "zoomed-in" version of the 128-GB experiment, focusing on the exact point in which the first transaction is executed for both ARIES and instant restart. As expected, instant restart accepts the first transaction after six seconds (not easily visible in the chart), in contrast with the 610 seconds of ARIES.

Figure 17 shows the average transaction latency during restart for the same experiment. The y-axis shows average latency from 0.1 to 1000 ms in a log-scale. During the offline period, no transactions are executed and so the latency value is plotted as zero. As observed in the throughput measurements, ARIES restart incurs longer offline phases the longer the redo length is. The warm-up curve, i.e., the progression of average latency from the maximum value to the steady-state value, on the other hand, is the same regardless of redo length. Note that steady-state latency is fairly low at ~ 0.3 ms—this is due to the non-durable commit tweak mentioned earlier. In a production system with an effective group commit implementation, steady-state latency would be noticeably higher while throughput would be similar to the results observed here.

For the next experiment, skew is introduced by directing 90% of the transactions to 10% of the TPC-C warehouses, while the remaining 10% of transactions are distributed uniformly. As discussed in Section 3.1.5, skew in the database access pattern does not result in skew in the log access pattern. This is because log records are only accessed once during redo recovery, i.e., there is no temporal

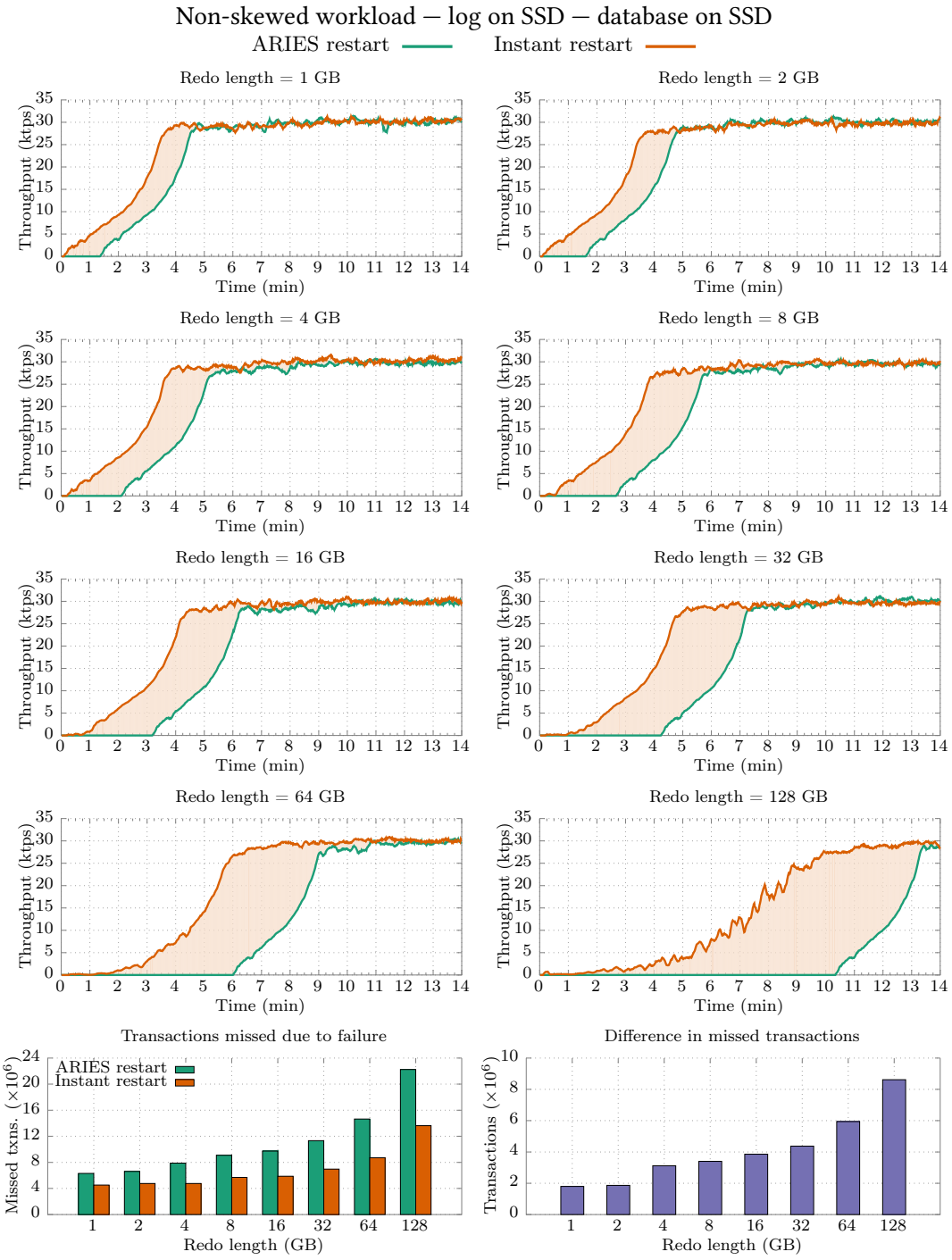

Fig. 15. Restart performance on non-skewed workload, with log and database on SSD

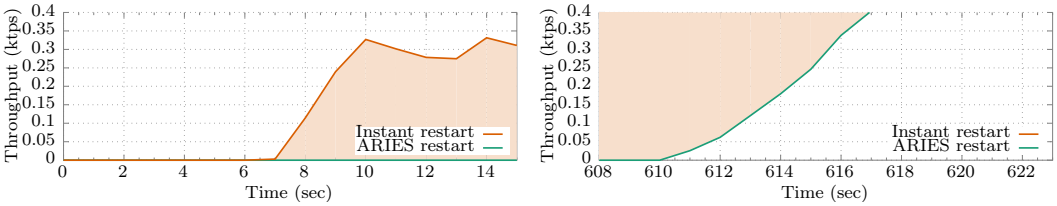

Fig. 16. Zoomed-in versions of the 128-GB experiment of Figure 15

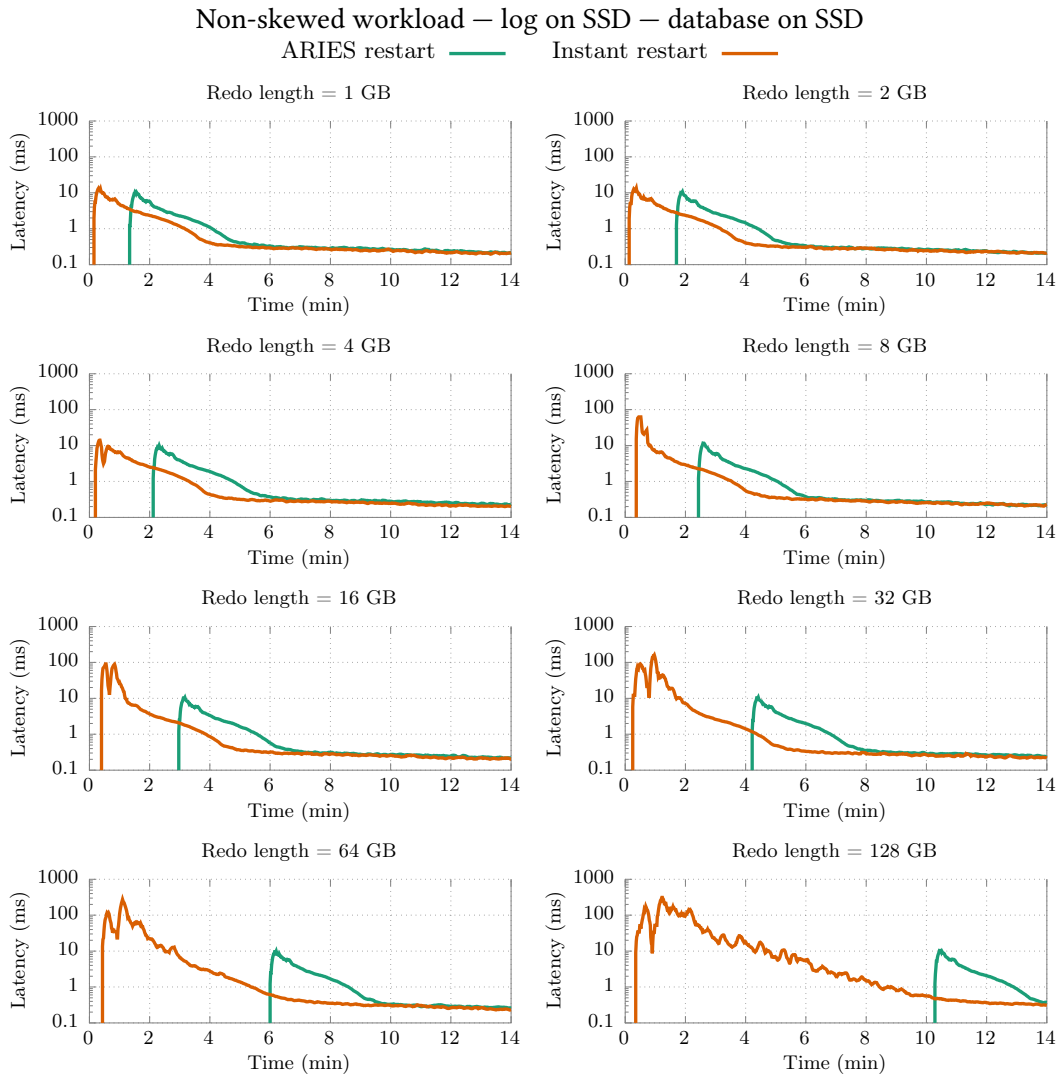

Fig. 17. Transaction latency after restart

locality, and all log records pertaining to dirty pages are accessed. However, the warm-up phase should be shorter for both ARIES and instant restart, since many more transactions can be executed with less data recovered.

The results for the skewed workload are shown in Figure 18. As the logistic curves show, warm-up is indeed faster, but there is no significant improvement for instant restart in comparison with the non-skewed workload—the difference in missed transactions is roughly the same. The conclusion here is that skew makes for faster restart for both ARIES and instant restart.

3.2.3 Log in RAM and database on SSD

The next experiment evaluates restart performance with the log in main memory and the database on SSD. Maintaining the log in volatile memory is obviously not practical, because it does not provide ACID guarantees, but this scenario aims to investigate the effects that can be expected from future NVM devices on restart performance. The latency of future NVM devices is expected to be higher than that of DRAM, and thus this experiment can be seen as an extrapolation to an ideal scenario, where the latencies are actually very similar.

Figure 19 shows the results, based on the skewed workload as discussed earlier. The first and most important conclusion is that instant restart performance is practically unaffected by the redo length, with practically equal logistic curves across all experiments. Furthermore, the steady-state transaction throughput is, as expected, much higher than with the log on SSD. A cause for the less regular behavior of the steady-state throughput could not be determined; as the plots show, it occurs for both ARIES and instant restart and it is thus not related to recovery.

In this scenario of extreme low-latency log devices, instant restart has a tremendous advantage over ARIES, because the duration of the offline and warm-up phases is largely independent of the amount of recovery work that is performed. As the bar charts on the bottom of Figure 19 show, the number of missed transactions remains constant in instant restart, while it grows linearly with the redo length in ARIES; in the 128-GB experiment, ARIES misses approximately 45 million transactions, roughly five times more than instant restart.

The conclusion of this experiment is that instant restart can immensely benefit from future NVM devices, while ARIES does not show any visible improvement in recovery performance. This is largely due to the fact that random I/O on the database device remains the major bottleneck, and improvements on log access speed are thus mostly irrelevant.

3.2.4 Log in RAM and database on HDD

A third experiment setup maintains the log in RAM to simulate NVM like earlier, but the database is stored on a hard-disk drive. This could become a realistic scenario in the near future, if the following predictions hold: (i) NVM devices become cost-effective enough to replace SSDs as a log device; and (ii) HDDs keep increasing in density, which translates into higher capacity and sequential bandwidth. In this envisioned scenario, there might be no favorable use for SSDs.

In terms of recovery efficiency, the main problem with this scenario is that redo recovery incurs random page reads on the database. Since, as a rule of thumb, read latency of HDDs is one-hundred times higher than SSDs, the warm-up phase would last about one-hundred times longer.

The results of this experiment in Figure 20 confirm this expectation, since after six hours the system has warmed-up to only 2 ktps—less than 3% of the maximum steady-state throughput. Given this long duration, the experiment was interrupted and executed only for the redo lengths of 1 and 16 GB. However, the collected data leads to similar conclusions as the previous experiment with

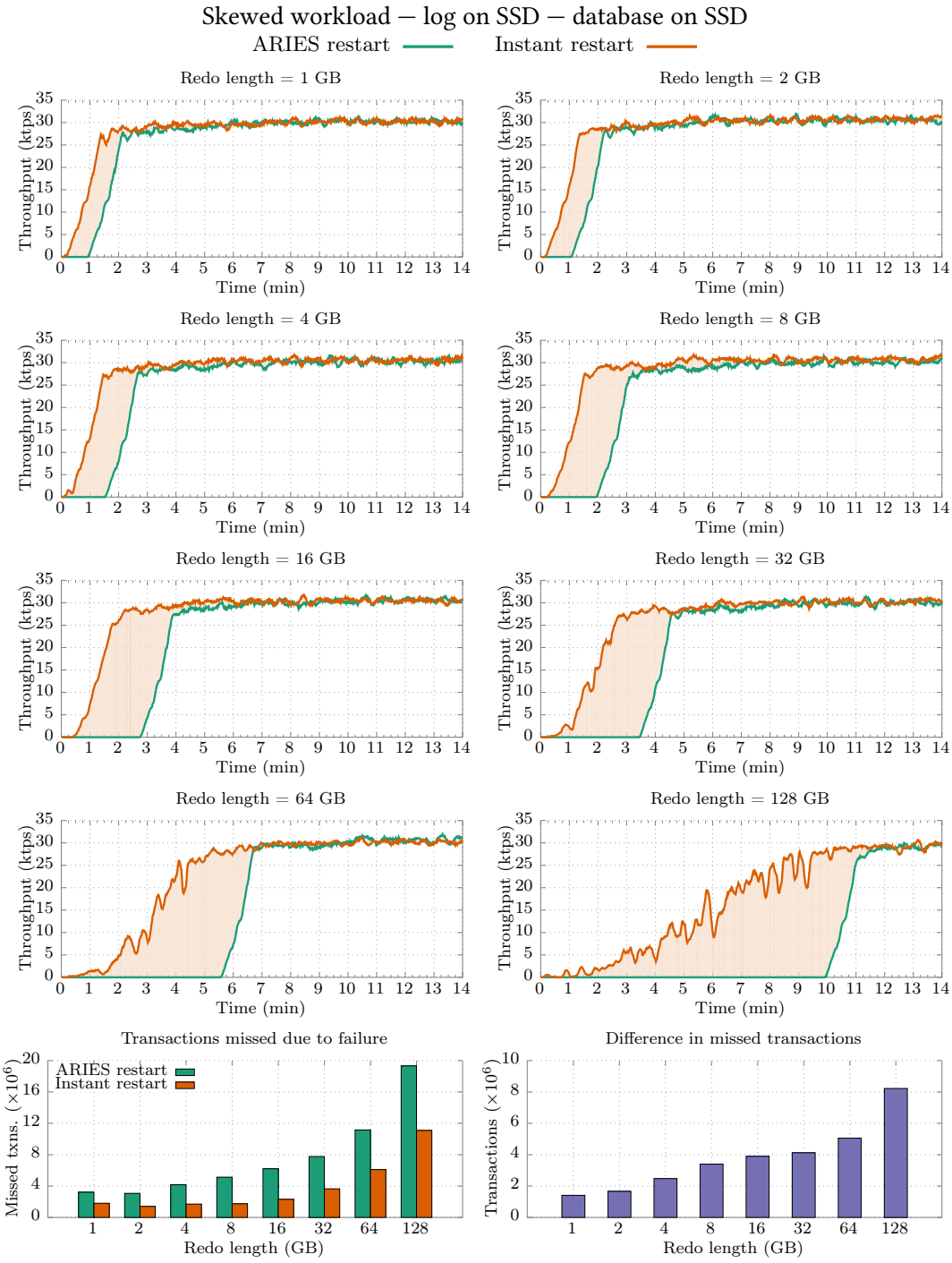

Fig. 18. Restart performance on skewed workload, with log and database on SSD

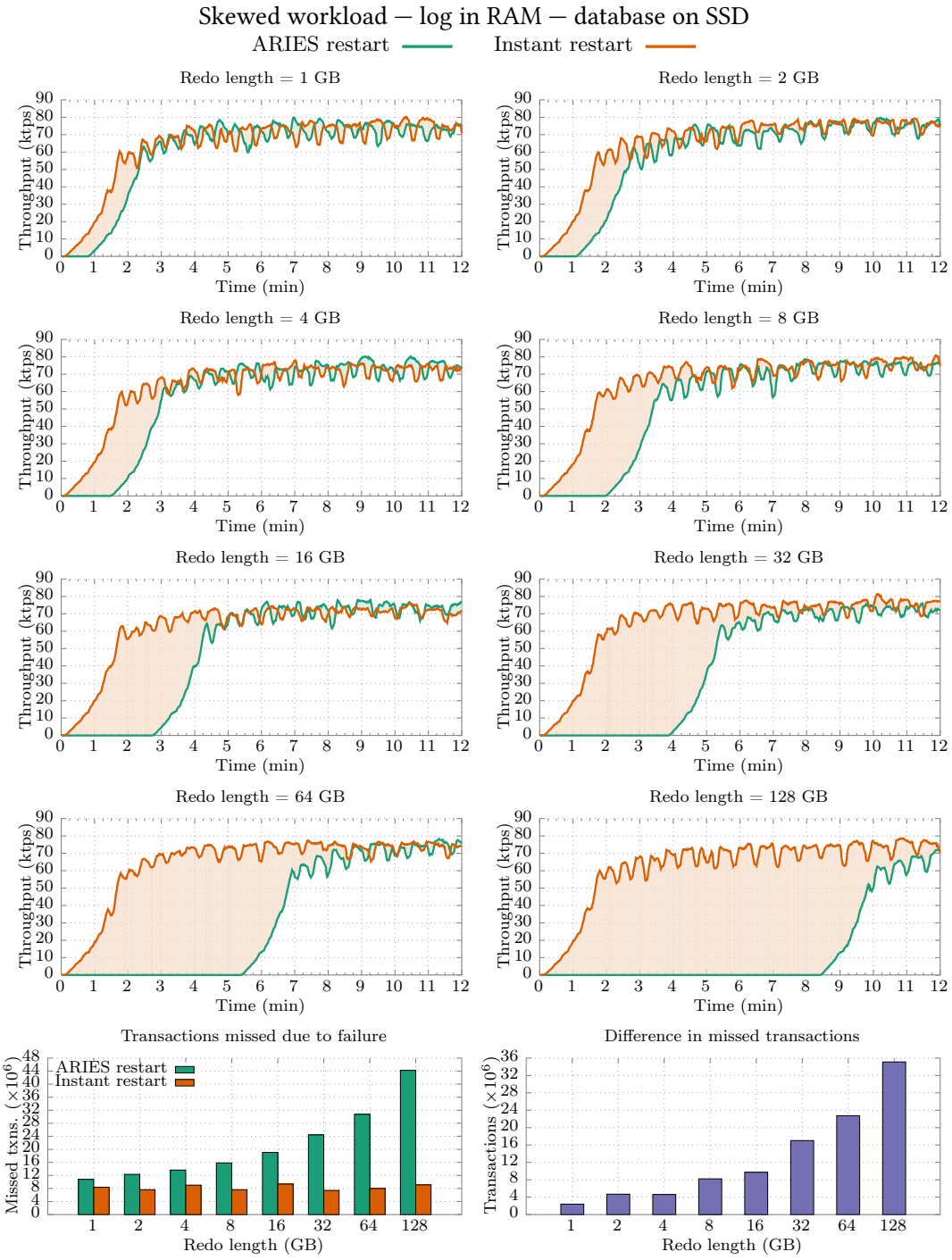

Fig. 19. Restart performance of skewed workload, with log in RAM and database on SSD

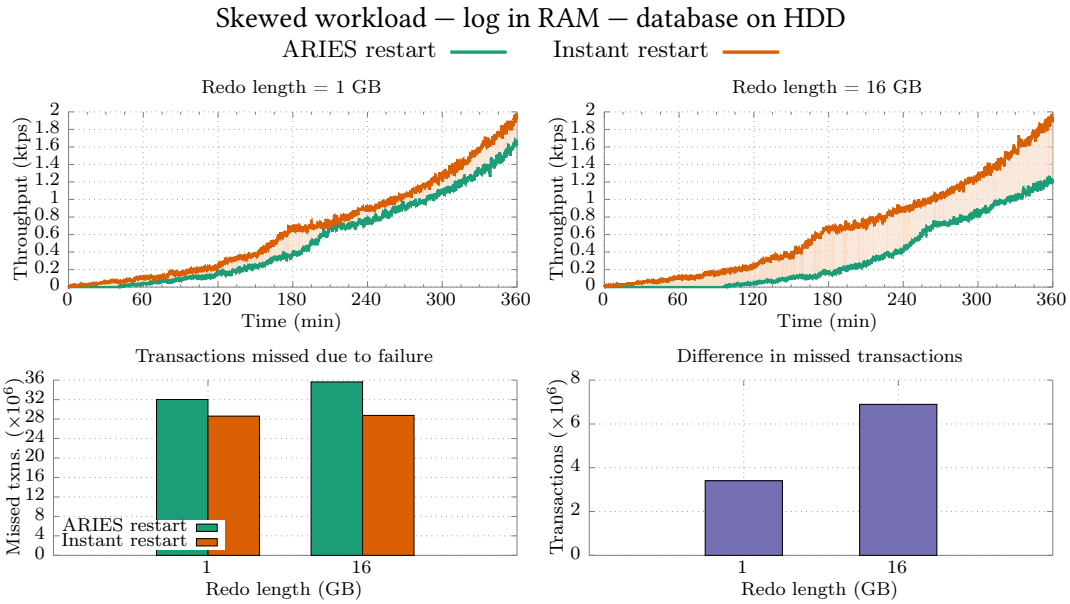

Fig. 20. Restart performance on skewed workload, with log in RAM and database on HDD

database on SSD, namely that warm-up in instant restart is independent of the redo length, while in ARIES there is a linear correlation.

In this experiment where the time scale is multiple hours rather than minutes, availability is an even bigger concern. In the 16-GB experiment, ARIES restart only accepts the first transactions after ~ 90 minutes, while instant restart still only needs about 5 seconds. In this case, the improvement on perceived availability is almost of three orders of magnitude.

Admittedly, such slow warm-up performance would be impractical in production systems, rendering the assumptions about the obsolescence of SSDs invalid. However, it must be pointed out that the prototype system used in these experiments is not optimized for database storage on HDDs. One crucial technique to speed up warm-up is to read multiple adjacent pages with one single read [Moh95, PDTP16], which is not implemented. Furthermore, log optimizations such as page-image log records can be used to substantially reduce the number of page reads required for recovery. While this optimization is implemented in the prototype, it was not utilized as a means to avoid database reads.

The conclusion of the HDD experiment is that the envisioned scenario could become common in practice if proper techniques to avoid random database reads during warm-up are implemented. In that case, instant restart is a valid alternative to ARIES, since it can exploit efficient random reads on the log to make warm-up time independent of the amount of recovery work and also to avoid random reads on the database. In many of the techniques presented in the remainder of this thesis, the log plays a much more active role, while the database tends to serve as an archive device, making the envisioned scenario more plausible.

3.2.5 Impact of concurrent I/O

Instant restart enables the execution of page-oriented redo and transaction-oriented undo recovery concurrently with the execution of new transactions. In addition to that, traditional, log-based

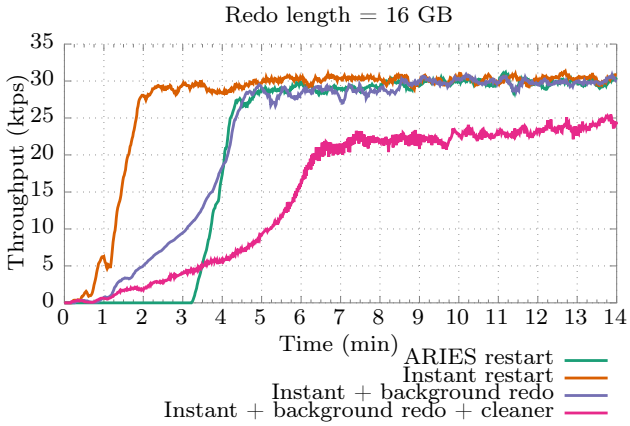

Fig. 21. Impact of background I/O activities on restart performance

recovery as performed in ARIES can also be carried out concurrently, as well as database I/O operations such as read-ahead and background page cleaning. However, these concurrent actions unavoidably impact restart performance, as measured by the steepness of the logistic curve. Therefore, such actions must be carefully scheduled.

To demonstrate how badly concurrent I/O can hurt recovery performance, the 16-GB experiment of Figure 15 (non-skewed workload; log and database on SSD) was repeated with two additional configurations: first, instant restart with concurrent, batch redo recovery on the background; and second, instant restart with both batch redo and aggressive page cleaning on the background. As the results, shown in Figure 21, clearly demonstrate, such aggressive background operations can make instant perform worse than ARIES without background activity. If background page cleaning is added to ARIES recovery, similar performance degradations should occur.

The conclusion of this experiment is not that ARIES can be a better choice than instant restart with concurrent I/O operations, because the impact of such operations on ARIES restart is also significant (although not measured here). Rather, the conclusion is that background I/O activity can significantly slow down recovery and warm-up performance, and thus it must be either delayed until the system reaches steady state or scheduled with low priority. Other examples of these activities include backups, log archiving, maintenance operations such as index rebuild or defragmentation, consistency checks, garbage collection, etc.

In all other experiments performed in this thesis, background activities such as page cleaning and batch recovery are delayed until the system reaches steady state.

3.3 Sorting and indexing log records

One of the key innovations of instant restart is the ability to recover each dirty page independently during redo recovery. This is achieved by retrieving the history of updates of an arbitrary page using the per-page log chain. In addition to the chain of log records, the LSN of the most recent update on that page must be provided using an auxiliary data structure. As shown in Figure 7, this data structure can be either the page recovery index or the dirty page table, depending on the type of failure from which the system is recovering. In both cases, it is apparent that these data structures provide a form of index on the log, enabling retrieval of log records by page ID.

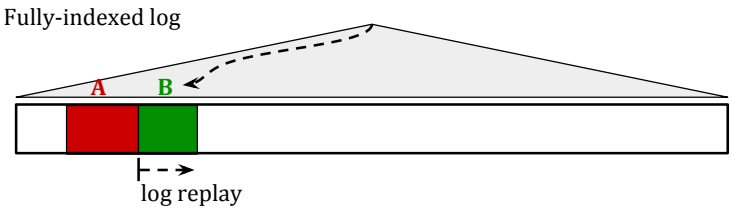

Fig. 22. Fully-indexed log: an ideal, but impractical data structure for log replay

This section explores techniques to reorganize the log by sorting and indexing, such that the process of retrieving log records by page ID is made more efficient. In the context of instant recovery, this new organization of the log permits sequential log replay for both restart and restore. In Chapters 4 and 5, novel techniques for update propagation, recovery, and data retrieval are proposed based on the new organization.

Sorting and indexing log records was first introduced for log archiving in previous work on media recovery [SGH15, SGH17, GGS16], which will be discussed in Section 3.4. However, the remainder of this thesis employs these techniques on a broader context; therefore, this section discusses the general problem of sorting and indexing the log, regardless of the type of failure and recovery to be supported.

3.3.1 Candidate data structures for log indexing

As discussed in Section 3.1.5, instant restart actually has a less efficient I/O pattern in the log than ARIES restart, because it performs random reads, whereas ARIES scans the log sequentially. The random log reads are required to fetch the history of log records pertaining to a given page by following the per-page log chain. Despite the random access pattern, which results in a slower warm-up, instant restart outperforms ARIES because of its significantly shorter offline phase. Furthermore, the inefficiency of random log reads can be largely mitigated with fast devices such as SSDs and enough main memory to cache the portion of the log relevant for redo, i.e., the redo length. Nevertheless, the fact remains that log replay can be improved substantially with less memory consumption if it can be performed more sequentially, i.e., with an increased degree of spatial locality in the log.

As Figure 22 illustrates, the ideal log organization for log replay efficiency would be a fully-indexed log, in which log records pertaining to a given page are always physically clustered. In this organization, only one random read is required to recover an arbitrary page, assuming that the inner nodes of the index are cached in main memory.

The problem with a fully-indexed log is that its maintenance would require random writes, in order to insert new log records in the correct sorted position. This, in turn, completely defeats one of the main purposes of a write-ahead log, which is to provide durability with sequential writes rather than random ones, following a no-force policy. Even if random log writes can be as efficient as log appends, guaranteeing the consistency of the log, i.e., making sure that persisted log records are never overwritten, is also more challenging and leads to higher fragmentation with random writes.

Therefore, a more practical implementation of a log index must provide an acceptable trade-off between query and update performance—in other words, it must retain the append-only characteristic of updates but allow reorganization steps to introduce some degree of sortedness to answer queries, i.e., the retrieval of a page's history, more efficiently. This is precisely the goal of *log-structured merge trees* (LSM trees), first proposed by O'Neil et al. [OCGO96]. They were introduced to support efficient

queries on time-ordered data, e.g., retrieve orders of a given customer from a sales history table. Recently, LSM trees have been widely adopted in "no-SQL" databases used primarily in cloud-based applications and social networks [CDG+08]. These systems must provide very fast data ingestion rates into very large databases while still providing indexing and querying capabilities, thus the goal of balancing query and update performance.

Despite providing the desired trade-off between read and write performance, LSM trees are not directly applicable as data structure to index log records, because updates are performed in-place in a memory-resident index (O'Neil et al. refer to this as the C_0 component [OCGO96]) which is made durable using a write-ahead log. Only when this component fills up the reserved main-memory storage, it is written into persistent storage, which enables log space recycling. For the purposes of efficient log replay during recovery, the data structure to be indexed is the log itself. Thus, it must be persisted atomically and immediately during commit processing. Relying on a separate log for durability, i.e., a "log of a log", completely defeats the purpose of indexing the log in the first place, in an almost paradoxical way[13].

In general, the requirement to persist log records at commit time precludes any technique that buffers updates in main memory, deferring their propagation until a single large write can be performed. A survey by Graefe [Gra06a] discusses several such techniques in a broader context than LSM trees. One way to reuse such update-buffering data structures for log indexing is by decoupling durability from the log index data structure—i.e., if commit processing relies on a traditional write-ahead log while queries (i.e., log replay) rely on the index. However, this makes it cumbersome to ensure transactional consistency and correct recovery, since these two data structures—write-ahead log and log index—must be coordinated carefully. Therefore, an appropriate log index data structure should provide both durability and efficient querying capabilities in a unified, coherent design.

Another key distinction between LSM trees and an indexed log is their usage pattern. LSM trees are typically used to directly store and retrieve user data, i.e., as a primary store for keys and values in the application's domain. An indexed log, on the other hand, maintains log records that describe changes to some other primary data store, and these must be indexed by the page identifier of this primary store. Furthermore, the value associated with each key is a list of log records that must retain their LSN order within the same page ID—i.e., the *history* of that page. Therefore, new log records can only be appended to these lists, rather than overwrite a previous value.

Lastly, the indexed log must reorganize log records without completely losing track of the LSN order, or some other form of database-wise version number such as epochs [TZK+13]. This is because LSNs are used for two purposes: to recycle log space and to control the progress of incremental indexing. Appended log records are gradually reorganized into indexed components, and the easiest way to manage this process is by keeping track of the LSN ranges associated with different portions of the log index. Furthermore, the indexed log should also maintain the low-water marks discussed in previous sections to support correct recovery and log space recycling.

In summary, an appropriate index structure for log records should satisfy the following requirements:

(1) Log records must be appended in a durable way as dictated by the commit protocol, without any form of internal buffering or write deferral.
(2) The index must support efficient retrieval of all log records pertaining to a given page ID, retaining the order of updates on each page.

[13] Like a "*mirror mirroring a mirror*" (Douglas F. Hofstadter)

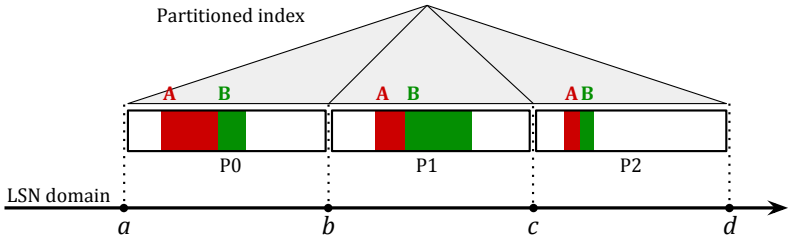

Fig. 23. Log index organized into partitions mapped to non-overlapping LSN intervals

(3) Indexing should be incremental, i.e., there must be a gradual and continuous transition from the append order into the fully-indexed order.
(4) The index must be organized into storage units that can be mapped to LSN ranges, in order to control the progress of incremental indexing and enable log space recycling with low-water marks.

With these requirements in mind and based on earlier research [OCGO96, Gra03a, Gra06a], the following section discusses data structures for log-record indexing.

3.3.2 *Partitioned log index*

This thesis proposes a log index data structure based on *partitions*. Before presenting details of the data structure implementation, this section first focuses on the general structure of a partitioned log index. This is illustrated in Figure 23, which shows a log index with three partitions—P0, P1, and P2. These are mapped to non-overlapping intervals of the LSN domain, illustrated as the left-to-right arrow on the bottom part of the diagram. In this case, all log records in the interval $[a, b)$ are stored in partition P0, ordered by page ID and, within the same page ID, by LSN. The same holds for the interval $[b, c)$ and P1 as well as $[c, d)$ and P2.

In the example of Figure 23, if page A is fetched from the database and it requires log replay, all its log records with LSN between a and d could be fetched from the partitioned index, without the need for a per-page log chain or a separate data structure (e.g., dirty page table) to provide the page's last LSN within the $[a, d)$ range. This log replay example assumes that log records with LSN greater than d are either nonexistent or not required for correct recovery—a requirement that will be discussed in detail below. Another advantage of the partitioned log index is that, depending on the number of partitions that must be inspected, the log-record access pattern is mostly sequential and permits simple read-ahead.

Using this basic log index mechanism, new partitions are created in bulk, by periodically scanning a portion of the recovery log, sorting it in a main-memory workspace, and appending it as a new partition on the partitioned index. Therefore, the log always has a *sorted* and an *unsorted* component, which are separated by an LSN value known as the *sort low-water mark*—in the example above, this is the value d. In Section 3.4, this sorting process is an integral part of *log archiving*, i.e., copying the recovery log into an archive device to support media recovery.

In order to support restart recovery with a partitioned log index, two requirements must be satifsfied:

(1) The unsorted portion of the log must be retained for as long as needed to support log analysis and undo recovery; these never access the partitioned log index.

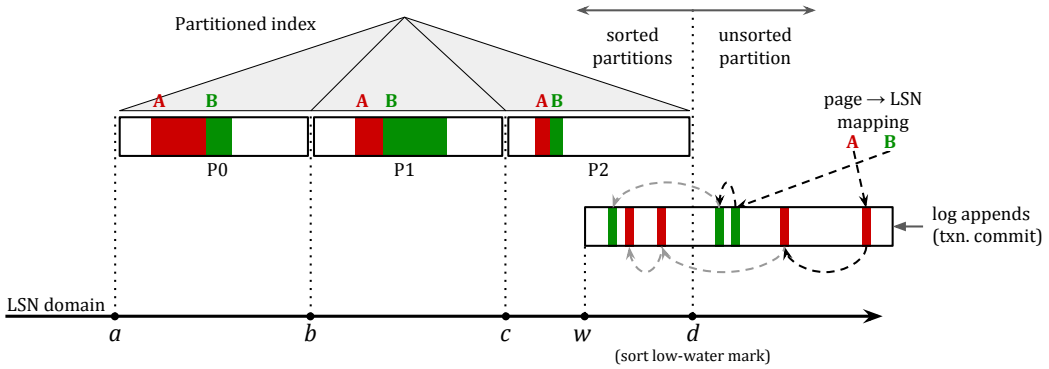

Fig. 24. Management of sorted and unsorted partitions within the log index

(2) Log replay during redo recovery must correctly keep track of the sort low-water mark, to ensure that all required log records are replayed exactly once and in the correct order.

Figure 24 illustrates the management of sorted and unsorted components. The sorted partitions are the same as shown before in Figure 23, but the diagram has been extended with the recovery log, which has restart low-water mark w—an LSN value between c and d. The portion of the recovery log from d to the log tail is the *unsorted partition*. As the diagram shows, log records are appended directly into the unsorted partition in LSN order, and the per-page log chain is maintained as described earlier.

Log analysis and undo recovery both make use of the unsorted log only. Therefore, the recovery log may not be recycled as soon as a new sorted partition is produced; instead, recycling is guided by the restart low-water mark (w in the example above). However, this water mark is now given by the minimum of the undo water mark and the latest complete checkpoint, ignoring the redo low-water mark. This is because redo recovery makes use of the sorted partition for log replay, as described below.

In order to recover a given dirty page during redo recovery, the first step is the same as described earlier for instant restart: the dirty page table (shown in the diagram more generally as a "page → LSN mapping") is consulted to retrieve the expected page LSN. That value is used as begin point for the per-page log chain traversal. With a partitioned log index, if a pointer in the chain contains an LSN lower than the sort low-water mark (d in the diagram), the traversal is terminated and the remaining log records are fetched from the sorted partitions. In the example of the diagram, recovering page A would fetch the latest two log records from the unsorted partition, whereas the remaining ones would be retrieved from the sorted partitions. If the recovery LSN of the page (not shown in the diagram) is higher than c, then only partition P2 must be probed. Further details of how look-up is performed on the partitioned log index are provided in Section 3.3.6 below.

An alternative design for the partitioned log index eliminates the unsorted partition, as shown in Figure 25. Here, a group commit protocol sorts and appends whole log pages at once, and each log page creates a new partition. Each partition can, in turn, define an "epoch", which is useful for resource management and concurrency control [TZK+13]. This design could be simpler to implement, given that it makes use of a single data structure, e.g., a partitioned B-tree [Gra03a]. However, it presents a trade-off in terms of log replay efficiency: probing possibly thousands of such

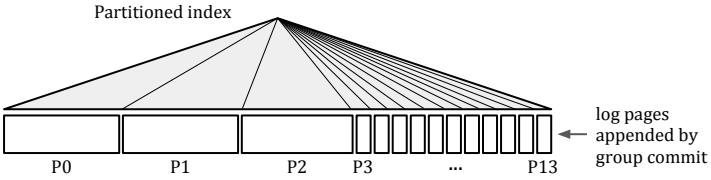

Fig. 25. A log index using just sorted partitions

small partitions might result in less efficient log replay as just traversing the per-page log chain on the unsorted partition. Furthermore, because persisted log records are not ordered primarily by LSN, the per-transaction log chain is lost, and undo recovery cannot be performed on a per-transaction basis. For these reasons, this alternative design could be better suited to no-steal recovery schemes, such as the approach presented in Chapter 5, where data structures for log indexing will be revisited. The remainder of the present chapter assumes an unsorted partition as discussed earlier.

Next, details of how to implement the partitioned log index with sorted and unsorted components are provided.

3.3.3 Index implementation

Several implementation alternatives exist for the partitioned log index. A partitioned B-tree [Gra03a], for instance, is a good candidate data structure, since it was designed for similar use cases. It can also be applied to provide fast updates at the expense of query performance [Gra06a], similar to LSM trees. However, because the intended use here is as a log data structure, durability must be provided by means other than WAL. For that end, the shadowing approach proposed by Rodeh could be employed [Rod08] as an atomic propagation scheme.

While a partitioned B-tree enables some degree of code reuse, because it can be built upon an existing B-tree implementation, extending it with an atomic propagation scheme might restrict reusability or introduce undesired complexity. For that reason, this section proposes a simpler implementation that relies on file- and operating-system support. Furthermore, it provides a great degree of flexibility, enabling simple and effective optimizations that will be discussed later on.

The implementation proposed here is called a *flat-file index*. The key idea is to store each partition as a file where log records are stored contiguously, without any explicit page organization. Access to each file is provided with memory mapping, e.g., through the *mmap* system call [GLIa]. Therefore, storage and buffer management are handled transparently by the operating system. Since partitions are created in bulk and never modified, typical propagation concerns that preclude the use of memory mapping in database systems (e.g., observing the WAL rule and keeping track of recovery LSNs) are not an issue here.

Index information is appended to each file as a simple list of `(pageID, offset)` pairs. It yields the offset of the first log record of a given page ID, and it contains only the page IDs for which log records exist in the file. Therefore, when looking up log records of a certain page ID, a random read on the file can be suppressed if that page ID is not found in the list. This has a substantial impact on log replay performance, since it eliminates reads of false-positive probes with 100% accuracy and no need for additional Bloom filters (which are common in LSM trees [OCGO96]). In the measurements performed in this thesis, the index size is always less than 0.5% of the file size; thus, storage consumption is not a concern.

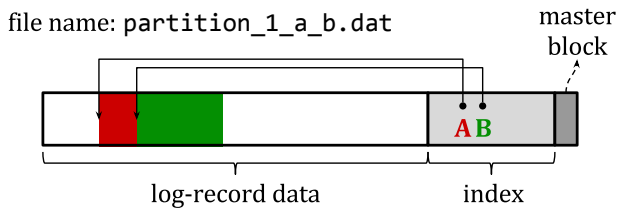

Fig. 26. Flat-file format of the partitioned log index

Partitions are identified by a three-component ID: the *level* of that partition, which will be discussed in Section 3.3.5 below, the *begin LSN*, and the *end LSN*. The latter two indicate the LSN range covered by that partition, as illustrated with the values *a*–*d* in Figure 24. These three components are stored as the file name. Thus, when the system is initialized, the set of available log partitions can be derived by simply listing the directory's contents.

Lastly, a *master block* stored in a predetermined offset of the file (e.g., the last 4 KB) can be used to store metadata information. For the design presented here, the minimal information required in this block is the offset on the file on which log-record data ends and the index begins.

Figure 26 illustrates the file format described above. The name of the file indicates that it corresponds to a level-1 partition covering the LSN range from *a* to *b*.

3.3.4 Run generation

The process of creating sorted partitions from the unsorted log is essentially the same as run generation for external merge-sort, and all implementation techniques and optimizations apply [Gra06b]. This section describes how run generation is implemented to generate log index partitions in the prototype used in this thesis.

In order to consume log records from the recovery log and produce them into the sorted partition output in a continuous manner, the replacement selection algorithm [Knu98] is used. Since log records are of variable length, replacement selection requires memory management of the sort workspace, which is implemented with the algorithm by Larson [Lar03].

In order to maintain the mapping of partitions to non-overlapping LSN ranges, the replacement selection algorithm must be adapted to not produce runs larger than the main-memory sort workspace. This is achieved by keeping track of a binary "color" for input and output, as shown in the example of Figure 27. The example shows three states that occur throughout the run generation cycle, from top to bottom. Nodes in the selection tree, which correspond each to a log record consumed from the recovery log, are annotated with either black or white. In the steady state, i.e., after the selection tree is full for the first time, input and output always have opposing colors. The comparison function of the selection tree must sort the output color lower than the input color. In the top state of Figure 27, a white partition is being produced, and thus all incoming entries are colored black. As soon as the top entry of the selection tree is a black one, all white inputs were consumed, and the current output partition is closed and renamed. At this point, shown in the second state in the diagram, input and output colors are reversed, and the comparison function now sorts black lower than white. In the third state, a black partition is being generated, which is the reverse of the first state. Using this mechanism, partitions are guaranteed to contain log records of non-overlapping, adjacent LSN ranges.

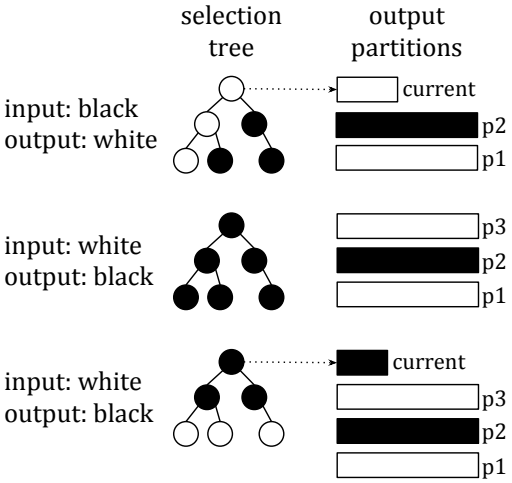

Fig. 27. Replacement selection algorithm using black and white colors to keep track of input LSN ranges

As the current partition is written out in the process described above, an auxiliary data structure keeps track of the page IDs and offsets produced. When a partition is closed, this information is appended into the file, producing the index discussed earlier. Once the index and the master block are successfully written out, the partition is finalized, and it can now be renamed, which essentially "commits" the addition of a new partition to the log index. This relies on the atomicity guarantee of the rename operation provided by the file system [GLIb]. If a system failure occurs and restart finds a temporary partition which has not been fully produced, it is deleted and the run generation process restarts from the end LSN of the last complete partition.

Since the partitioned log index can only be used for log replay, i.e., for redo recovery, log records can be compressed by eliminating undo information such as before-images of records, transaction identifiers, the `undo_next` and `prev_txn` pointers, as well as the `store_id` field, which is only required for logical compensation. Further index compression techniques can also be applied [Gra06b, Gra11], thus reducing the size of the log index even further. Such compression opportunities were not exploited for the experiments in this thesis.

3.3.5 Merging partitions

The log indexing scheme presented so far exhibits unbounded growth on the number of partitions, which deteriorates the query performance—i.e., log replay performance—over time. To avoid that, partitions have to be *merged* in order to produce larger partitions, which cover larger LSN ranges and thus reduce the number of index lookups required to retrieve log records of a given page. For that same reason, merging is one of the essential techniques behind LSM trees [OCGO96].

The *level* component of the partition identifier is essential to keep track of merged partitions while guaranteeing the consistency of log replay—i.e., guaranteeing that no log record is missed and no log record is retrieved more than once. When a partition is first created by the run generation process discussed above, it has level 1, and its identifier is denoted as $\langle 1, a, b\rangle$, where a is the begin LSN and b the end LSN. A merge of partitions of level i yields a partition $\langle i+1, x, y\rangle$, where x is the lowest begin LSN of the merged partitions and y is the highest end LSN. Only consecutive partitions

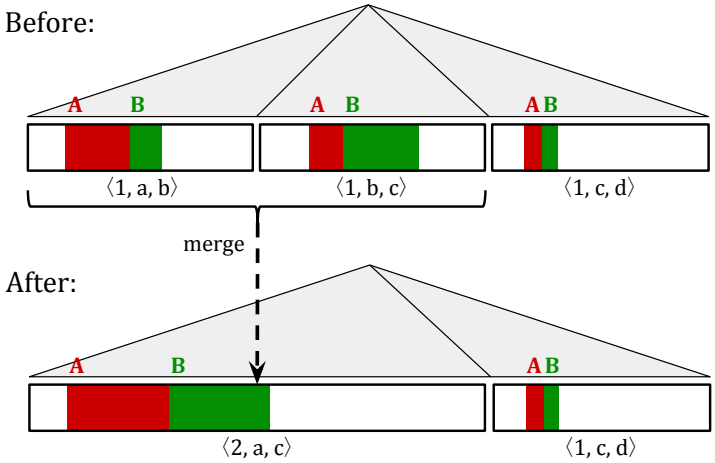

Fig. 28. Merging partitions of the log index

may be merged, so that all log records in the $[x, y)$ range are guaranteed to be contained in the new partition.

Figure 28 illustrates the merge process, based on the same partitioned log example seen so far. Here, the labels P0–P3 are replaced by the actual three-component identifiers. A merge operations produces a new partition $\langle 2, a, c\rangle$ from the inputs $\langle 1, a, b\rangle$ and $\langle 1, b, c\rangle$. The new index state, shown in the bottom, only requires two lookups instead of three to retrieve the history of pages A or B. In practice, a much higher merge fan-in would be used, so that the number of lookups is reduced even further.

As lower-level partitions are merged into higher-level partitions, the former can be recycled, freeing up space on the log device. Recycling cannot be performed eagerly because concurrent queries on the log index might fail or read corrupted data if partitions are deleted. Instead, standard garbage collection techniques such as reference counting should be employed. The prototype implemented in this thesis keeps track of handle objects for each partition, which includes the operating-system file descriptor as well as a reference counter. When a partition is picked for recycling, its reference counter is simply decremented, and actual deletion is deferred until the counter reaches zero. Because deleting multiple files is not an atomic operation, additional measures are required to handle system failures. One option is to implement partition deletion as a system transaction, which is very similar to other bulk deletions such as an index drop. Another option is to proactively remove during restart all lower-level partitions whose LSN ranges overlap with a higher-level one.

Merging also presents opportunities for compression, thus reducing overall space consumption even further. Applicable compression techniques include not only standard techniques for index compression [Gra11], but also the generation of page-image log records. These can be used to establish an upper bound on the storage consumption of the indexed log; for instance, a page-image log record can be generated whenever the log volume for a particular page surpasses the size of a database page itself. This kind of compression presents a trade-off between storage consumption and preservation of history—if the application must keep a record of every single update on the database, then page-image log records cannot be used to compress log partitions.

Merging improves lookup performance and compresses the log, but that comes at a cost on two important factors: write performance, because it consumes valuable bandwidth that could potentially be used to commit more transactions per unit of time; and also write amplification, as the same information is essentially rewritten on every merge step. Providing an acceptable trade-off between these factors is one of the key challenges driving research on LSM trees and similar indexing techniques [SR12, DAI17]. Investigating such trade-offs is out of the scope of this thesis, but most techniques proposed for LSM trees also apply for merging policies in the partitioned log index.

3.3.6 Log index lookup

This section explains how to perform lookups in the partitioned log index in order to retrieve the history of log records pertaining to a given page. The lookup algorithm is also able to deliver history for a set of contiguous pages, which is useful for the restore techniques discussed in Section 3.4.

The lookup algorithm of the partitioned log index requires additional logic—in comparison with a typical (non-partitioned) index—to handle multiple partitions with possibly overlapping LSN ranges. Furthermore, partitions can be merged to form new partitions and deleted anytime during a lookup and also while the cursor returned by a lookup is being iterated.

The partitioned log index resembles adaptive indexing techniques based on partitioned B-trees [GK10], and thus concurrency control techniques can be borrowed [GHI+12]. However, the access pattern is different from that of adaptive secondary indexes, and thus lookups with concurrent merges and deletions can be implemented in a simpler way. Unlike adaptive secondary indexes, partitions of the log index are read-only, i.e., they do not permit updates and queries do not move key ranges to different partitions. Therefore, threads can freely access log partitions without any concurrency control, as long as partitions are not deleted prematurely, which can be avoided with proper garbage collection.

Since concurrent merges may introduce higher-level partitions with overlapping LSN ranges, each thread accessing the log index must first atomically capture a snapshot of the list of partitions currently available. Therefore, a minimal amount of concurrency control is required for this preliminary step. The approach presented here relies on a simple in-memory data structure called the *partition catalog*—in some ways, it resembles the "table of contents" data structure used in adaptive merging [GK10]. Access to this data structure is protected with a global read-write latch.

The partition catalog keeps track of a list of partition identifiers (i.e., their level, begin LSN, and end LSN) organized by level, from the highest to the lowest. This is illustrated in Figure 29, which shows partitions of different levels with overlapping LSN ranges. When performing a log index lookup, the first step is to derive a list of partitions covering the desired LSN range without any gap or overlap. Since higher-level partitions require fewer random reads, they should have preference over lower-level partitions. In the example of Figure 29, a lookup uses the shaded partitions, which cover the entire LSN range shown with at most five random reads.

Algorithm 3 shows a possible implementation of a probe procedure for the partition catalog called *ProbePartitions*. Its arguments are a page ID range given by *beginPID* and *endPID* as well as an LSN range given by *beginLSN* and *endLSN*. As mentioned earlier, the partitioned log index supports log replay on a set of consecutive pages, which is why a page ID range is given. The LSN range restricts the set of partitions to be probed using knowledge from the caller's context. For example, redo recovery during restart might have both the recovery LSN and the expected page LSN; these two values establish lower and upper bounds for the LSN of the log records required to recover that given page. If no LSN range is given, all partitions available in the log index are considered.

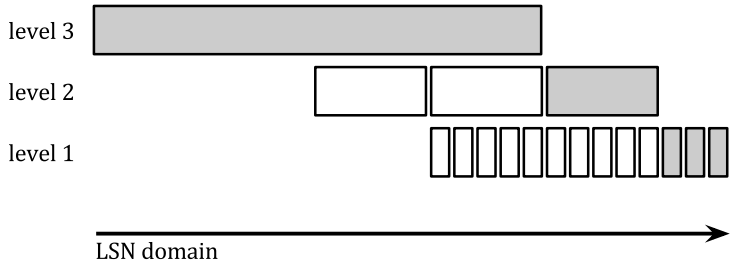

Fig. 29. Partition catalog showing partitions used in a lookup (shaded)

The algorithm collects partitions by covering the LSN range from begin to end, starting with the highest level available. It relies on the catalog providing a *maxLevel* function, that yields the highest level among all partitions, as well as a *getPartition* function that retrieves a pointer to a partition given its level and an LSN value inside that partition. A *currLSN* value keeps track of the LSN range covered so far; it starts with the given *beginLSN* and is updated with the end LSN of each partition retrieved using *getPartition*. Whenever no partitions are found for a given level and *currLSN*, the probe continues on the next lower level, until all levels are covered.

Line 16 illustrates the use of a simple filter, which is any kind of aggregate information about the distribution of values inside a partition. Here, each partition keeps track of the maximum page ID (*maxPID*) contained in it. If the maximum page ID in a probed partition is lower than the given *beginPID*, that partition is not included in the result list. This simple filter is used here for illustration purposes; in practice, filters can include any kind of materialized aggregate function and also Bloom filters [Gra09a].

If a partition passes this first filtering step, binary search is performed in the actual index, i.e., the list of (`page ID`, `offset`) pairs. If the given page ID range is not found, the entry returned by the binary search will contain a higher page ID, which gives another opportunity to filter out partitions that do not contain relevant log records. This type of filter is very effective in practice, especially if a single page is being recovered, since in that case it provides 100% accuracy—i.e., all random reads in the file yield log records that will be used in log replay.

Finally, after the partitions passes all filters and the file offset where log records will be read from is determined, a file handle f is opened in line 23 and a `PartitionInput` object is created and added to the result list. This object is used by the next algorithm as a log-record iterator on a single partition file.

Algorithm 4 shows the three basic functions used to scan the partitioned log index, following the traditional open-next-close interface of the iterator model [Gra93]. The first function, *OpenLogIndexScan*, probes the partition catalog as described earlier and calls the *open* method of each `PartitionInput` iterator object. This method, which is abstracted here, initializes the internal state of the iterator by reading the first log record. Partitions are opened from last to first, i.e., in reverse LSN order. This enables the algorithm to exploit page-image log records. In this case, earlier partitions in the LSN range can be excluded from the log replay, because the page state can be fully reconstructed by replaying only the log records from this partition onwards. However, this optimization only works for single-page queries—i.e., if $beginPID = endPID - 1$—because a page-image log record is not necessarily present for all pages being recovered.

Once the *OpenLogIndexScan* procedure has probed the partition catalog and opened the required `PartitionInput` iterators, it builds a priority queue from these iterators in line 13. This queue is

Algorithm 3 Algorithm to probe the partition catalog when querying the log index

```
1:  procedure ProbePartitions(beginPID, endPID, beginLSN, endLSN)
2:      result ← ∅
3:      currLSN ← beginLSN
4:      pCatalog.latch.acquire(SHARED)
5:      level ← pCatalog.maxLevel()
6:      while level > 0 do
7:          if currLSN >= endLSN then
8:              break
9:          end if
10:         p ← pCatalog.getPartition(level, currLSN)
11:         if p = NULL then
12:             level ← level − 1
13:             continue
14:         end if
15:         currLSN ← p.endLSN
16:         if beginPID > p.maxPID then
17:             continue
18:         end if
19:         entry ← binarySearch(p.entryList, beginPID)
20:         if entry.PID >= endPID then
21:             continue
22:         end if
23:         f ← openFile(p.file)
24:         result.append(new PartitionInput(f, entry.offset))
25:     end while
26:     pCatalog.latch.release()
27:     return result
28: end procedure
```

used to merge the partitions and deliver a single log stream sorted by page ID; thus, it is returned to the caller as an iterator object for the log index scan. Note that, for single-page queries, the priority queue is not necessary, since partitions input can simply be consumed from first to last. However, this optimization is omitted here.

The *NextLogRecord* procedure is used to consume log records from the priority queue. The comparison function used to build the priority queue and reorganize it for every consumed log record is not shown here, but it is assumed that the `PartitionInput` object gives access to the page ID and LSN of its current log record—these two values are used by the comparison function to sort the log record output stream. Finally, once the log index scan is consumed by the caller, the *CloseLogIndexScan* procedure is invoked to close whatever remaining `PartitionInput` objects that have not been fully consumed.

The algorithms described here permit page-oriented log replay from the partitioned log index. The next section describes how to use this mechanism for redo recovery during instant restart. After that, techniques for media recovery based on the partitioned log index will be introduced.

Algorithm 4 Partitioned log index scan

```
1:  procedure OpenLogIndexScan(beginPID, endPID, beginLSN, endLSN)
2:      partitions ← ProbePartitions(beginPID, endPID, beginLSN, endLSN)
3:      size ← partitions.size()
4:      i ← size − 1
5:      while i >= 0 do
6:          partitions[i].open()
7:          lr ← partitions[i].getLogRecord()
8:          i ← i − 1
9:          if beginPID = endPID − 1 and lr.type = page_image then
10:             break
11:         end if
12:     end while
13:     pq ← MakePriorityQueue(partitions[i + 1 : size])
14:     return pq
15: end procedure

16: procedure NextLogRecord(pq)
17:     if pq.empty() then
18:         return NULL
19:     end if
20:     top ← pq.top()
21:     if not top.finished() then
22:         lr ← top.getLogRecord()
23:         top.next()
24:         pq.reorganize()
25:         return lr
26:     else
27:         top.close()
28:         pq.removeTop()
29:         return NextLogRecord(pq)
30:     end if
31: end procedure

32: procedure CloseLogIndexScan(pq)
33:     while not pq.empty() do
34:         pq.top().close()
35:         pq.removeTop()
36:     end while
37: end procedure
```

3.3.7 Restart with partitioned log index

The partitioned log index can be used to support log replay in the instant restart algorithm. This was illustrated and briefly discussed earlier with Figure 24. When collecting the log records to be replayed to recover a given page, the per-page log chain in the unsorted partition is traversed, as in

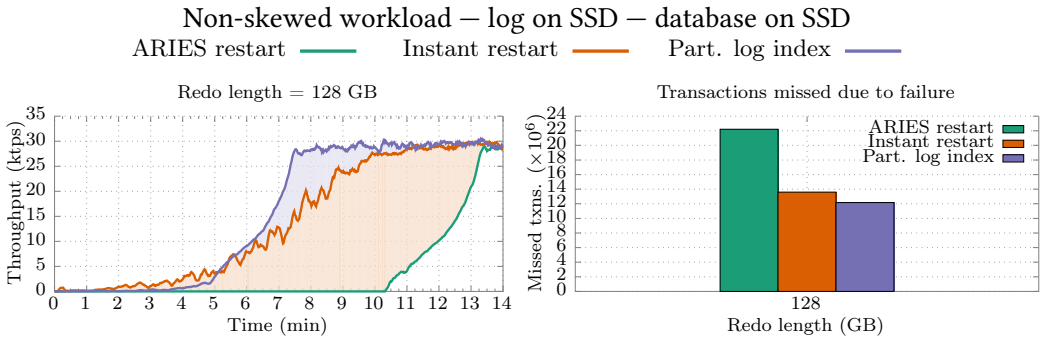

Fig. 30. Instant restart performance with partitioned log index

normal instant restart. But, as soon as the chain points to an LSN lower than the sort low-water mark, the remaining log records can be fetched using the log index iterator described previously. For that, the *OpenLogIndexScan* procedure is called with the ID of the given page as *beginPID* and the successor page ID as *endPID* (since *endPID* determines an exclusive interval). As *beginLSN*, two options are available: if log analysis keeps track of both the expected page LSN and the recovery LSN of each dirty page, the latter is used; otherwise, if the page is fetched from the database beforehand, its current page LSN is passed. As *endLSN*, the LSN of the last log record collected from the per-page log chain—or the expected page LSN, if it is lower than the sort low-water mark—is passed.

Once the log scan is opened, it is iterated using the *NextLogRecord* procedure and all log records are applied on the page. When the iterator finishes, the log records collected during traversal of the per-page log chain, if any, are applied, as in normal instant restart.

The experiment of Figure 30 analyzes restart efficiency using the partitioned log index, using the non-skewed workload, with both log and database on SSDs—i.e., the experiment of Figure 15. The experiment uses the dataset with 128-GB redo length, of which 100 GB are in sorted partitions; these are organized into one level-two partition of 52 GB and five level-two partitions of 7 GB. Note that the total size of sorted partitions (87 GB) is less than the 100-GB log range that they cover, thanks to compression with page-image log records.

As the experiment shows, the partitioned log index exhibits a steeper warm-up curve, so that the number of missed transactions is reduced from ~14 to ~12 million, in comparison with instant restart. This is not a significant improvement, but is shows the potential for more efficient log replay. While sorting and indexing the log might not be justifiable for restart purposes alone, given the small improvement, it provides substantial benefits for media recovery, as discussed in Section 3.4 below. Later on, Chapters 4 and 5 present novel techniques for transactional storage and update propagation based on the partitioned log index.

3.4 Instant restore

Advancements in hardware technology have significantly improved the performance of database systems over the last decade, allowing for throughput in the order of thousands of transactions per second and data volumes in the order of petabytes. Availability, on the other hand, has not seen drastic improvements, and the research goal postulated by Jim Gray in his ACM Turing Award Lecture of a system "unavailable for less than one second per hundred years" [Gra03b] remains an open challenge. Improvements in reliable hardware and data center technology have contributed significantly to the availability goal, but proper software techniques are required to not only avoid

failures but also repair failed systems as quickly as possible. This is especially relevant given that a significant share of failures is caused by human errors and unpredictable defects in software and firmware, which are immune to hardware improvements [Gra86, P+02]. In the context of database logging and recovery, the state of the art has unfortunately not changed much since the early 90's, and no significant advancements were achieved in the software front towards the availability goal.

Instant restore is a technique for media recovery that drastically reduces mean time to repair with simple software techniques. Like instant restart, its main advantage is enabling on-demand, independent recovery of fine-granular objects. Instant restore relies on the partitioned log index introduced earlier to address the limitations of ARIES restore, which were briefly discussed in Section 2.8. Before describing the restore algorithm itself, this section briefly discusses media failures, focusing on their causes and effects.

3.4.1 Media failures

A media failure is detected when the operating system delivers an error code upon a page read or write, or when an inconsistency is detected on a page after it is read. For instance, inconsistencies can be detected on individual pages when traversing a B-tree index [GKS12a]. If single-page repair [GK12] is supported, the system must decide if the detected failure applies to the whole device or simply to the page just accessed. One option is to rely on diagnostics provided by the device drivers. Alternatively, all I/O failures could be treated as a single-page failure initially and later on escalated to a whole device failure (e.g., if a corrupted page is repaired with single-page recovery but another failure occurs when writing it back to a different location).

Three steps are required when initializing the restore process: (i) provisioning of a replacement device and preparation of (ii) backup devices and (iii) the log archive. As discussed in Section 2.8, restore basically works by loading an old database image from backups, then replaying the log archive on it, and gradually save the restored pages into a replacement device, which also serves read operations for already-restored pages.

For the first step—provisioning of a replacement device—a formatted and empty stand-by device is ideally available to avoid mechanical or human delays. To maintain quality of service after restore is completed, it is also advisable to maintain identical stand-by and active devices, whereby one stand-by device can serve multiple active ones. These operational concerns are out of the scope of this thesis; therefore, the discussion herein as well as the empirical evaluation later on both assume that the replacement device is immediately available.

The second step—loading an old database image from backups—is the first main distinction between instant restore and ARIES restore. In ARIES, this is a separate, offline stage that cannot generally be interleaved with log replay or transaction execution. Instant restore, on the other hand, loads individual *segments* of the database—i.e., fixed-length, contiguous sets of pages—on demand, as explained in detail later on. One key requirement of such on-demand loading is that, if the backups (both full and incremental) are compressed (e.g., by skipping unallocated database pages and by suppressing white space), they must be indexed to allow direct access to individual segments.

Every backup file, both full and incremental, is associated with a *restore low-water mark*. This is the LSN value up to which all updates are guaranteed to have been propagated into that backup (and all previous backups until the latest full backup). In ARIES nomenclature, this LSN is called the "media recovery redo point" [MHL+92]. It serves as a lower bound for the log replay phase.

Lastly, the log archive must also be prepared for access. Instant restore maintains the log archive as sorted partitions of the partitioned log index, and thus it must be ready for access before recovery begins. The instant restore algorithm presented here assumes that all log records that will be required

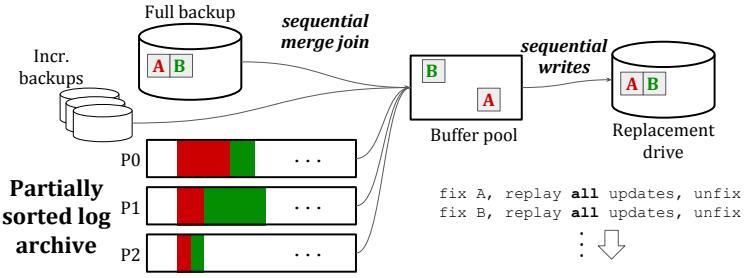

Fig. 31. Single-pass restore

for restoring the failed volume are propagated from the active transaction log, i.e., the unsorted partition, into the log archive, i.e., the sorted partitions. To that end, a *restore_begin* log record is generated as soon as the media failure is detected—its LSN is the *restore high-water mark*. Then, the log archiving process is invoked synchronously to load all log records up until this mark into sorted partitions.

An alternative implementation to synchronously invoking the log archival process is to make use of the partitioned log index lookup algorithm described in Section 3.3.6, which can replay log records on a page-by-page fashion using both the sorted log archive partitions and the unsorted recovery log. One caveat is that it requires a page recovery index or embedded *child LSN* values on parent-to-child pointers [GGS16] to determine the expected page LSN, i.e., the entry point of the per-page log chain traversal.

When the provisioning process is completed, the restore manager initializes its in-memory data structures and begins the restore process, which is discussed in the remainder of this section.

3.4.2 Single-pass restore

Single-pass restore [SGH15, GGS16] is a predecessor technique of instant restore. It addresses the problem of random database reads during log replay, which is one of the main deficiencies of the ARIES restore algorithm, discussed in Section 2.8. In single-pass restore, the replacement device is fully restored to its most recent state using a single sequential pass over log archive and backups. This is achieved by maintaining a *partially sorted log archive*, which is a data structure similar to the partitioned log index discussed earlier, but without the actual index on each partition and thus only supporting full scans.

Figure 31 illustrates single-pass restore and its sequential access pattern. Partitions of the log archive are sorted by page ID. During log replay, these partitions are merged to produce a single sorted log stream. This is stream is further merged with pages coming from the backups—again in page ID order. If incremental backups are available, they must be merged with the latest full backup, so that only the most recent version of each page is delivered. This process permits page-at-a-time restoration of the entire device. As pages come in from the backups, their matching log records are applied; this pattern is essentially a merge join between the sorted log stream and backup pages.

With a fully sequential access pattern, single-pass restore is able to restore an up-to-date database in about the same time it takes to copy a full backup in traditional restore techniques, as shown in the experiment on the left side of Figure 32. This experiment (for which details are available in the original publication [SGH15]) measures the time it takes to copy databases of different sizes

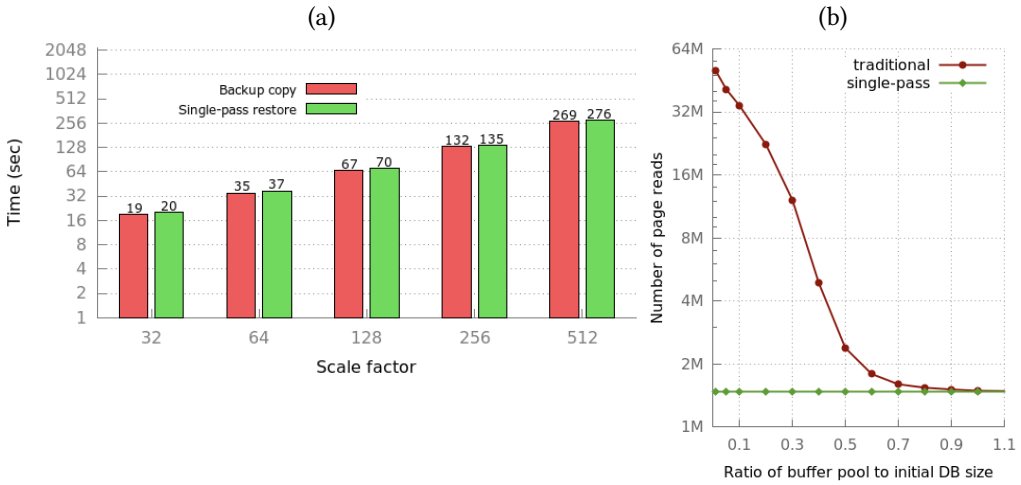

Fig. 32. (a) Time to perform single-pass restore vs. copy a full backup; (b) Number of database reads as a function of buffer pool size for traditional log replay vs. single-pass restore.

(varying the TPC-C scale factor on the x-axis) versus the time it takes to perform log replay using the partially sorted log archive. As the results show, the time required by single-pass restore is within a marginal fraction of the time for copying the full backup. This marginal difference is attributed to the restoration of new pages which were allocated after the backup was taken.

Another advantage of the sequential access pattern is that is consumes very little memory. In traditional ARIES log reply, database pages are not only read in random order, but also read multiple times depending on the buffer pool size. This is because pages can get evicted and later on re-fetched if more log records need to be replayed. This is demonstrated by the experiment on the right side of Figure 32 (also taken from the original publication [SGH15]). Here, the buffer pool size is varied on the x-axis as a fraction of the database size. For each size, 48 GB of log data is replayed using traditional, LSN-ordered log and the partially sorted log. As the results show, the number of page reads remains constant with single-pass restore, as it requires one read per page regardless of buffer pool size. With traditional log replay, on the other hand, the number of reads increases exponentially as the size of the buffer pool decreases.

Besides the significant improvement on log replay performance and thus on media recovery efficiency, single-pass restore has useful applications in the management of backups and replication [GGS16]. Most notably, it makes incremental backups obsolete, since their only purpose, as discussed in Section 2.8.2, is to shorten the length of log replay. If log replay is performed sequentially, its length is much less of a concern, and incremental backups can actually slow down the recovery process. Furthermore, the partially sorted log archive enables the generation of full backups by replaying the log on an older full backup—essentially performing single-pass restore to generate a new backup rather than recover a failed device. This form of backup, known as *virtual backup* [GGS16], makes the generation of full backups very efficient and simple to implement, without interfering with transaction processing on the active database. For all these reasons, the complexity of generating, managing, and restoring from incremental backups is hardly justified.

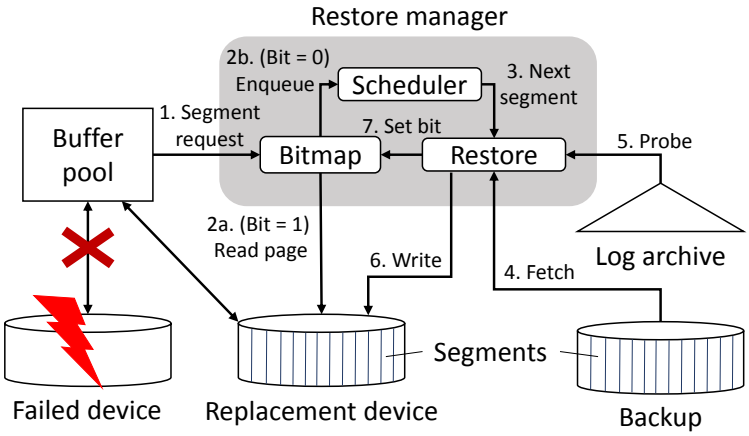

Fig. 33. Instant restore flow chart

3.4.3 *Instant restore algorithm*

The main goal of instant restore is to preserve the efficiency of single-pass restore while allowing more fine-granular restoration units (i.e., smaller than the whole device) that can be recovered incrementally and on demand. Instant restore can be seen as a generalized approach based on *segments*, which consist of contiguous sets of data pages. If a segment is chosen to be as large as a whole device, the algorithm behaves exactly like single-pass restore; on the other extreme, if a segment is chosen to be a single page, the algorithm behaves like single-page recovery [GK12]. As discussed in this section and evaluated empirically in Section 3.5, the optimal restore behavior lies somewhere between these two extremes, and simple adaptive techniques are proposed to robustly deliver good restore performance without turning knobs manually.

When a media failure is detected, a restore manager component is initialized and all page read and write requests from the buffer pool are intercepted by this component. The diagram in Figure 33 illustrates the interaction of the restore manager with the buffer pool and all persistent devices involved in the restore process: failed and replacement devices, log archive, and backup. For reasons discussed earlier, the present discussion does not consider incremental backups, since they make the algorithm simpler and do not provide any benefit in practice. Nevertheless, the algorithm can be easily adapted to support incremental backups as well.

In the following discussion, the numbers in parentheses refer to the numbered steps in Figure 33. The restore manager keeps track of which segments were already restored using a segment recovery bitmap, which is initialized with zeros. When a page access occurs, the restore manager first looks up its segment in the bitmap (1). If set to one, it indicates that the segment was already restored and can be accessed directly on the replacement device (2a). If set to zero, a segment restore request is enqueued into a restore scheduler (2b), which coordinates the restoration of individual segments (3).

To restore a given segment, an older version is first fetched from the backup directly (4). This is in contrast to ARIES restore, which first loads entire backups into the replacement device and then reads pages from there [MHL+92]. This has the implication that backups must reside on random-access devices (i.e., not on tape) and allow direct access to individual segments, which might require an index if backup images are compressed. These requirements, which are also present in single-page repair [GK12], seem quite reasonable given the very low cost per byte of current high-capacity hard

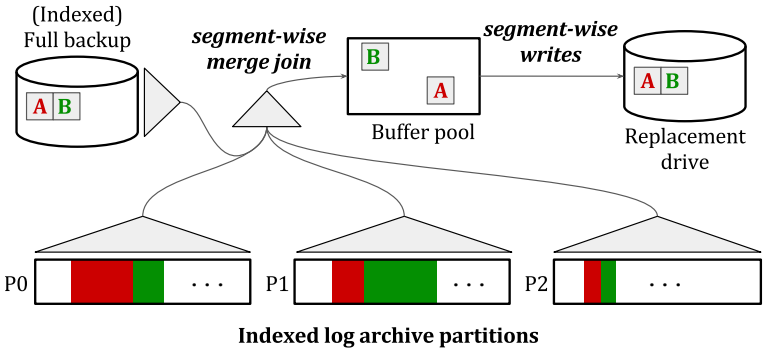

Fig. 34. Access pattern of instant restore

disks. For moderately-sized databases, it is even advisable to maintain log archive and backups on flash storage.

While the backed-up image of a segment is loaded, the indexed log archive data structure is probed for the log records pertaining to that segment (5). This process utilizes the algorithms discussed earlier in Section 3.3.6. If page-image log records are employed, it might be beneficial to delay reading backup segments until after the log is probed, since the presence of such log records avoids unnecessary reads from the backup. After that, log replay is performed to bring the segment to its most recent state, after which it can be written back into a replacement device (6).

Finally, once a segment is restored, the bitmap is updated (7) and all pending read and write requests can proceed. Typically, a requested page will remain in the buffer pool after its containing segment is restored, so that no additional read is required on the replacement device when a transaction accesses a restored page.

All read and write operations described above—log archive index probe, segment fetch, and segment write after restoration—happen asynchronously with minimal coordination. The read operations are essentially merged index scans—a very common pattern in query processing [Gra93]. Writing a restored segment to the replacement device can also be done asynchronously, preferably by simply reusing the buffer pool page cleaner.

To illustrate the access pattern of instant restore, Figure 34 shows an example scenario with three log archive partitions and two pages, A and B, belonging to the same segment. The main difference to the previous diagrams is the segment-wise, incremental access pattern, which delivers the efficiency of pure sequential access with the responsiveness of on-demand random reads.

Using this mechanism, user transactions accessing data either in the buffer pool or on segments already restored can execute without any additional delay, whereby the media failure goes completely unnoticed. Access to segments not yet restored are used to guide the restore process, triggering the restoration of individual segments on demand. As such, the time to repair observed by transactions accessing data not yet restored is multiple orders of magnitude lower than the time to repair the whole device. Furthermore, time to repair observed by an individual transaction is independent of the total capacity of the failed device. This is in contrast to previous methods, which require longer downtime for larger devices.

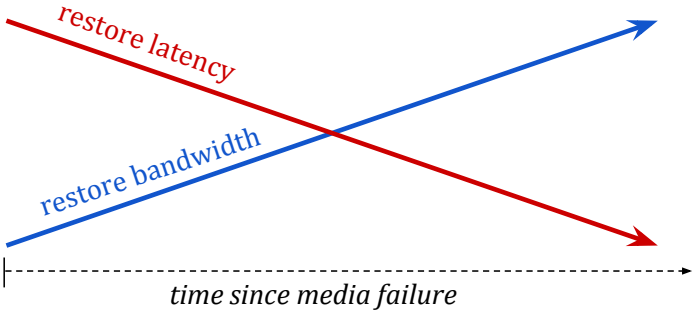

Fig. 35. Latency vs. bandwidth behavior during instant restore

3.4.4 Latency vs. bandwidth trade-off

One major contribution of instant restore is that it generalizes single-page repair and single-pass restore, providing a continuum of choices between the two. In order to optimize restore behavior, the restore manager must adaptively and robustly choose the best option within this continuum. In practice, this boils down to choosing the correct granularity of access to both backup and log archive, in order to balance restore latency and bandwidth.

Restore latency is defined as the additional delay imposed on the page reads and writes of an individual transaction due to restore operations. Hence, it follows that if a single page can be read and restored in the same time it takes to just read it, the restore latency is zero—this is the "gold standard" of restore performance and availability. For a single transaction, restore latency can be reduced by setting a small segment size—e.g., a single page. However, this is not the optimal behavior when considering average restore latency across all transactions. Therefore, restore bandwidth, i.e., the number of bytes restored per second, must also be optimized. The optimized restore behavior is illustrated in Fig. 35: in the beginning of the restore process, pages which are needed more urgently should be restored first, so that restore latency is decreased; towards the end, less and less transactions must wait for restore, so the system can effectively increase restore bandwidth while a low restore latency is maintained.

It is also worth noting that devices with low latency and inherent support for parallelism, e.g., solid-state drives, make these trade-offs less pronounced. This does not mean, however, that instant restore is any less significant for such devices—a point which is emphasized in the next two paragraphs.

As discussed earlier, previous restore techniques suffered from two deficiencies: inefficient access pattern and lack of incremental and on-demand recovery. Solid-state devices shorten the efficiency gap between restore algorithms with sequential and random access, but this gap will never be entirely closed, especially considering the locality and predictability of sequential access, and thus its proneness to prefetching.

As for the second deficiency, low-latency devices directly contribute to the reduction of restore latency, because the time to recover a single segment is reduced with faster access to backup and log archive partitions. Therefore, with instant restore, any improvement on I/O latency directly translates into lower time to repair—as perceived by a single transaction—and thus higher availability. Non-incremental techniques, where the restore latency is basically the time for complete recovery, do not benefit as much from low-latency storage hardware when it comes to improving restore latency.

In terms of latency and bandwidth trade-off in the instant restore algorithm, the first choice to be made is the segment size. In order to simplify the tracking of restore progress with a simple bitmap data structure, a fixed segment size must be chosen when initializing the restore manager. A good choice seems to be a size such that acceptable bandwidth is delivered even for purely random access, but not too many segments exist such that the bitmap would be too large—e.g., 2 MB for both SSD and HDD.

In order to exploit opportunities for increasing bandwidth, multiple contiguous segments should be restored in a single step when applicable. One technique to achieve that dynamically and adaptively is to simply run single-pass restore concurrently with instant restore. Since the two processes rely on the same algorithm, no additional code complexity is required. Furthermore, the coordination between them is essentially the same as that between concurrent instant restore processes—they both rely on the buffer pool and the segment recovery bitmap.

In terms of log archive access, the size of initial (i.e., level-one) partitions poses an important trade-off between minimizing merge effort and minimizing the lag between generating a log record and persisting it into the log archive. In order to generate larger partitions, log records must be kept longer in the in-memory sort workspace. On the other hand, correct recovery requires that all log records up to the time of device failure be properly archived before restore can begin; thus, smaller initial runs imply lower restore latency for the first post-failure transactions. One simple technique that can potentially mitigate this concern is to adapt instant restore to use the recovery log, i.e., the unsorted partition of the partitioned log index, as discussed earlier in Section 3.3.6. Instant restore could be used to bring all pages in a segment to the state up to the sort low-water mark. After that, single-page recovery using the per-page log chain could be applied to apply the remaining log records to each page of a segment. This should only be necessary for the first few segments, as the sort low-water mark should quickly reach the restore high-water mark; from that point on, restore operations can rely solely on the sorted partitions.

Besides these concerns specific to instant restore, established techniques to choose initial partition size and merge fan-in based on device characteristics directly apply [Gra93]. This is mainly because the access pattern of instant restore basically resembles that of an external sort followed by a merge join.

3.4.5 Coordination of multiple failures

As mentioned briefly above, the segment recovery bitmap enables the coordination of concurrent restore processes, allowing configurable scheduling policies. Another important aspect to be considered is the coordination among restore and the other recovery modes summarized in Section 2.1. This section discusses how to coordinate all such recovery actions without violating transactional consistency.

The first failure class—transaction failure—is the easiest to handle because its recovery is made transparent to the other classes thanks to rollback by logical compensation actions, as introduced in ARIES [MHL+92] and refined in the multi-level transaction model [WV02]. The implication is that recovery for the other failure classes must distinguish only between uncommitted and committed transactions. Transactions that abort are simply considered committed—it just happens that they revert all changes they made, i.e., they "commit nothing". Therefore, for the purposes of instant restore, transactions that issue an abort behave exactly like any other in-flight transaction, including those that started after the failure: they hold locks to protect their reads and writes and access data through the buffer pool, which possibly triggers segment restoration as described earlier.

Instant restart and single-page repair can be executed concurrently because they both perform log replay on a single page at a time, and thus coordination relies on the latching protocol of the buffer pool. If a single-page failure occurs during instant restart, it is detected in the fix operation, which invokes single-page recovery to bring that page to its most recent state. Therefore, no further action will be required for redo of instant restart, because it will detect that the page LSN matches the expected value registered in the dirty page table. If a single-page failure happens after restart redo has recovered it, then single-page recovery is simply invoked as it would normally be—regardless of ongoing restart recovery on other pages.

If a system failure happens during single-page recovery, the failure will be detected again during restart, because the fix call on that page will result in the same failure. Therefore, single-page recovery is simply re-invoked. Alternatively, a system transaction can be used to register the fact that a single-page failure was detected, along with the new storage location of the recovered page—this is similar to page migration in write-optimized B-trees [Gra04]. Upon restart, the single-page recovery process is simply resumed, and coordination works exactly as described above.

While instant restart and single-page recovery both work at a page granularity and can thus be coordinated using latches in the buffer pool, that is not enough for instant restore. Here, a segment, whose size is fixed when a failure is detected, is the unit of recovery, and coordination relies on the segment recovery bitmap. Using two states—restored and not restored—avoids restoring a segment more than once in sequence, but additional measures are required to prevent that from happening concurrently. One option is to simply employ a map with three states, the additional one being simply "undergoing restore". A thread encountering the "not restored" state attempts to atomically change it to "undergoing restore": if it succeeds, it initiates the restore request for the segment in question; otherwise, it simply waits until the state changes to "restored".

Alternatively, coordination of segment restore requests can reuse the lock manager. A shared lock is acquired before verifying the bitmap state, and, in order to restore a segment, the shared lock must be upgraded to exclusive with an unconditional request. The thread that is granted the upgrade is then in charge of restoration, while the others will automatically wait and be awoken by the lock protocol, after which they see the "restored" state.

While the segment recovery bitmap provides coordination of concurrent restore processes, the buffer fix protocol is again used to coordinate restore with the other recovery modes. Concomitant restart and restore processes may occur in practice because some failures tend to cause related failures. A hardware fault, for instance, may not only corrupt persistent data, but also cause an operating system crash.

In order to not lose the progress of instant restore in case of a system failure, three restore actions must be logged: *begin*, *segment restore*, and *end*. During log analysis after a system failure, a begin log record causes the system to initialize the data structures of instant restore and redirect all page reads and writes to the restore manager, as described earlier. As segment-restore log records are found, the restore bitmap is updated accordingly, letting the system know which segments have already been restored prior to the system failure. In this case, a segment can only be considered fully restored once it has been fully written into the replacement device; thus, the segment-restore log record is only generated after that happens. Finally, if an end log record is found, the system knows that full restoration was completed, and it can return to normal mode.

Note that with a backward log analysis, as discussed in Section 3.1.2, this logic is reversed, and a completed restore is simply ignored because the end log record comes first. Furthermore, because log analysis only scans from or until the latest checkpoint, the state of the restore bitmap must be included in checkpoints as well.

After log analysis is completed, the restart and restore processes will be automatically coordinated with the methods described above. Restart recovery will fix pages in the buffer pool prior to performing any redo or undo action. The fix call, in turn, will issue a read request on the device. If the device has failed, the restore manager will intercept this request and follow the restore protocol described above. Only after the containing segment is restored, the fix call returns. After that, the page may still require log replay in the redo phase of restart, which is fine—the two recovery modes will simply replay different ranges of the page's history.

3.4.6 Implementation issues

The conceptual diagram of Figure 33 does not illustrate how restore interacts with other parts of the database system architecture, most importantly the buffer pool. Reusing the buffer pool component is highly advisable, as it provides several advantages. The following paragraphs discusses the interaction between the restore manager and the buffer pool in detail, emphasizing the advantages of reusing the buffer pool as well as highlighting some important concerns for a practical implementation.

Relying on the normal *fix* protocol to access pages and replay log records on them allows high concurrency between restore actions and transactions accessing pages already in the buffer pool, i.e., pages that don not require immediate on-demand restoration. Transactions accessing these pages do not observe any delay from restore actions. Furthermore, once the segments containing these pages are picked by the restore scheduler, log replay can be skipped on them, since they are already up to date.

Frames of the buffer pool should also be used as the restore workspace, serving as buffer into which pages are loaded from the backup and log records are replayed into. In order to exploit large reads and fetch whole segments at once, the buffer pool should provide a prefetch function that allocates multiple frames and reads pages from the backup with a single scatter-gather I/O call (e.g., `readv()` in Linux). Once the read is performed, each allocated frame should only be added to the buffer pool if that page is not yet cached (i.e., not found in the page-ID lookup table). This prefetch function is not exclusive to instant restore—regular prefetch functionality has the same logic and should therefore be reused. Lastly, because such prefetched pages are in an older state, a control block flag is used to indicate that they are still in need of recovery; user transactions that happen to fix them must therefore coordinate with the restore manager.

As segments are restored, the regular page cleaning protocol of the buffer pool takes care of writing them into the replacement device. Using proper I/O scheduling that can exploit large page writes (as discussed in detail in Chapter 4), whole segments—or even multiple adjacent segments—are written at once without any explicit involvement of the restore manager. As in the prefetch case, buffer pool functionality is reused with minimal restore-specific code. One crucial requirement in this case is that segment-restore log records, which are required to support recovery from system failures during restore, should only be produced after all pages of a segment have been successfully written out. Alternatively, log records can be generated immediately after log replay; in case of a concomitant system failure; log analysis must then additionally mark a segment as "unrestored" if any of its pages are in the dirty page table.

One important concern involving the synchronization of restore actions and user transactions is to avoid lost updates that can occur when restoring segments in the buffer pool. This can happen with the following sequence of events: (1) a page of the failed device is already in the buffer pool when the failure is detected; (2) when the restore manager is initialized, it waits for the `restore_begin` log record to be archived (as discussed earlier, this is required to make sure that all updates up to the

failure point are replayed during restore); (3) after that, the page is updated by a user transaction in LSN x, written into the replacement device, and subsequently evicted from the buffer pool; (4) now, the segment containing that page is picked for restore, which fetches the old version from the backup and replays all log records until the `restore_begin` LSN; (5) finally, that page is written by the cleaner and evicted. When this last write is performed, the update on LSN x is lost, because the restored version of the page was older than the one on the replacement device.

Unfortunately, there is no straight-forward solution for this lost-update problem. The key issue here is that neither the restore manager nor the buffer pool have knowledge of individual pages written by the page cleaner. Thus, the situation above can only be detected if system keeps track of individual pages restored rather than whole segments. In step 4 above, for example, an auxiliary data structure could be used to inform the restore manager that that particular page does not need recovery, or that it should be fetched from the replacement device rather than the backup. Such data structure would complement the segment recovery bitmap by keeping track of individual pages already restored in the segments that are not yet marked as restored in the segment recovery bitmap.

An alternative solution reuses the single-page recovery infrastructure. If a page recovery index is maintained, either as a separate data structure or embedded in parent-to-child page pointers [GK12], then the overwrite of step 5 below would be detected when the page is re-fetched, applying the necessary single-page recovery actions as needed. This mechanism also forgoes the need to wait for the log archiver to reach the `restore_begin` log record, since every page would be guaranteed to be fully recovered with a combination of log-archive replay and single-page recovery. Furthermore, it could be easily combined with the write elision technique [GGS16] to alleviate write pressure on the replacement device. Lastly, a middle-ground solution would be to wait for the log archiver as described earlier, but only create and maintain the page recovery index for the duration of instant restore; this achieves the same guarantees without having to maintain the page recovery index during normal processing. This last solution was implemented in the prototype of this thesis.

3.5 Restore experiments

This section presents an empirical evaluation of instant restore, focusing on the impact of restore on concurrent transactions as well as on the efficiency of restore itself, i.e., how quickly the failed media is fully restored. Unlike the experiments described earlier for instant restart, warm-up performance is not a concern here, because main-memory contents are not lost during a media failure. Rather, the goal is to investigate how much the media failure disturbs transaction processing, looking at both throughput and latency. Lastly, an experiment is also performed to measure the overhead of sorting and merging log records in the log archiving process.

3.5.1 Environment and workload

The workload considered in the following experiments is the same TPC-C benchmark used for the instant restart experiments in Section 3.2. However, it is run on a different server, whose configuration is shown in Table 3. These experiments also only consider SSD devices.

To generate the dataset for these experiments, a database of 10 GB is loaded (TPC-C scale factor 75) and copied to a separate file to serve as full backup. Then, the benchmark is executed until another 10 GB of log data is produced; the log is then archived into one level-3 partition of 5 GB, six level-2 partitions of 750 MB, and four level-1 partitions of 110 MB. Finally, the database is cleaned and the recovery log is emptied. During an experiment, the benchmark is executed on this dataset with 8 worker threads for five minutes to warm-up the buffer pool, after which a media failure is injected. During this initial phase, both the page cleaner and the log archiver run constantly in the

Processor	4 × Intel Xeon X5670 (12 cores total)
Main memory	100 GB DDR3 1333 MHz
Operating system	Ubuntu Linux 16.04 Kernel 4.4.0
Compiler	gcc 5.4 with -O3 optimization
SSD devices	Samsung 840 Pro 256 GB

Table 3. Server used for restart experiments

background. The experiment then continues until the complete device is restored and then for five more minutes after that. All experiments described below use the same dataset and follow this same pattern.

3.5.2 Segment size, latency, and bandwidth

The first experiment analyzes restore efficiency with different segment sizes. A buffer pool of 10 GB was chosen for the benchmark; this corresponds to the size of the initial database and it is not too small so that the process would be I/O-bound, but also not too large so that on-demand restore requests are very rare. The sizes analyzed are between 512 KB (64 pages) and 8 MB (4,000 pages), varying in exponential steps.

The first result, shown in Figure 36, shows the pattern of segment restoration over time for four different segment sizes. The x-axis shows the time since the beginning of the experiment, while the y-axis shows segment numbers (in increments of 1,000). Each dot in the chart represents the successful restoration of one segment. As the pattern shows, many segments are restored in a very scattered way in the beginning of the restore process; this is expected because, in this phase, many on-demand requests from transactions arrive in the restore scheduler. As time goes on, less requests arrive, making the pattern more sparse.

One visible feature in this experiment is the effect of the background restore thread, which performs single-pass restore in parallel but with low priority, i.e., it only picks up segments to restore if no requests exist in the restore scheduler. The effect of background single-pass restore is shown as the sequence of dots in the lower part of the charts, which at the very end rises to the top rapidly. The lower range of segment numbers contains important pages, such as catalogs and B-tree inner nodes—these are kept cached in the buffer pool, so that no restore request ever arrives. Thus, it is up to the background process to restore them. Towards the end of the experiment, the background restore process picks up the remaining segments, filling up the "holes" in the segment recovery bitmap. Once is passes the last segment, the replacement device has been fully restored and it may thus terminate the restore procedure. This restore pattern described above is observed in all segment sizes shown in Figure 36, but in different granularities. These charts also show that the smallest segment size (512 KB) requires about 12 minutes for complete restoration, while the largest one (8 MB) requires only 3 minutes.

The next result, based on the same experiment, is shown in Figure 37, which plots average restore bandwidth over time for each segment size. This chart shows an interesting result: in the first minutes of the restore process, bandwidth, i.e., the rate at which segments are restored, is very similar for all segment sizes. This is because, in this stage, all worker threads are performing restoration of the segments they request in parallel, so that the aggregate bandwidth does not depend so much on the segment size. This is largely due to the fact that the experiments use SSDs, which support a high degree of I/O parallelism. Towards the end phase of the restore process, most segments are restored

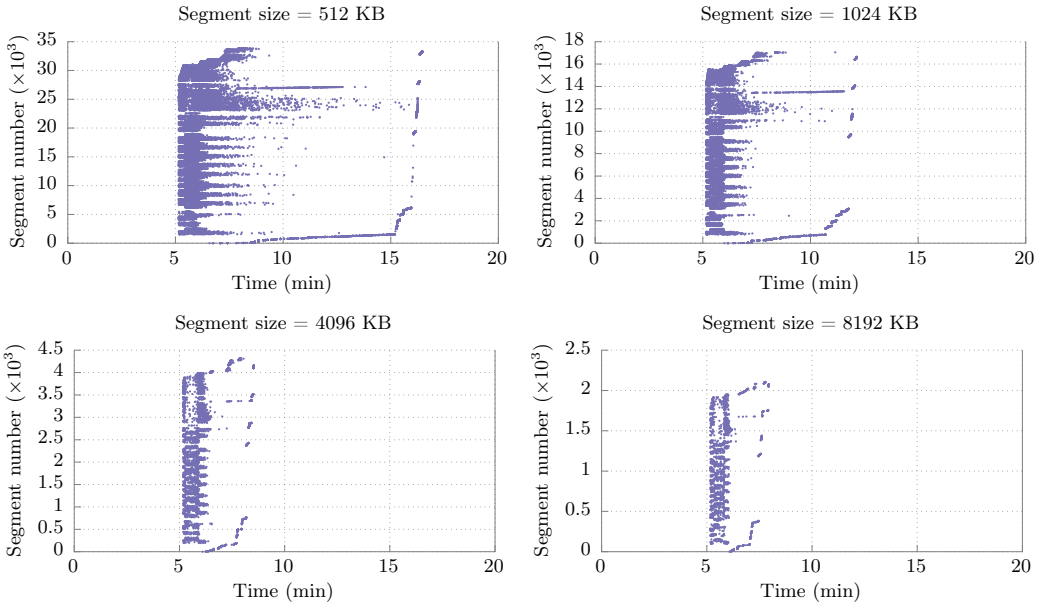

Fig. 36. Pattern of segment restoration during instant restore

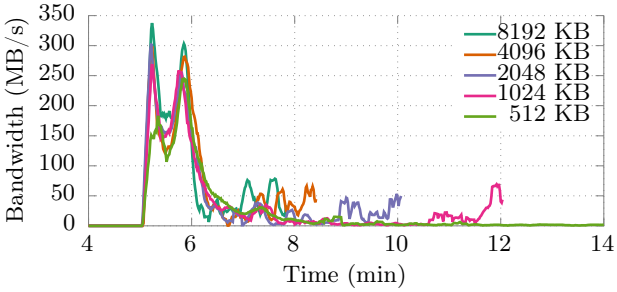

Fig. 37. Restore bandwidth over time with different segment sizes.

by the background single-pass procedure, and thus the observed bandwidth becomes much more dependent on the segment size. The bandwidth observed in the end phase is therefore the main factor determining the total restore time.

Figure 38 plots the average transaction latency during this same experiment in two different representations: on the left, a time series of average latency values and, on the right, the distribution of these values in a box plot. On the left chart, the expected behavior described earlier in Section 3.4.4 is observed: the smaller the segment, the less restore latency is incurred on transactions waiting to access data on the failed device. Note that the y-axis is in logarithmic scale; thus, during the first minute of media failure, average latency with 512-KB segments is one order of magnitude lower than with 8-MB segments. On the box plot on the right, 50% of the observed latency values fall within the boxes, while the lines at the bottom and at the top extend to the lowest and highest latency observed, respectively. The maximum value is due most importantly to the wait for the log

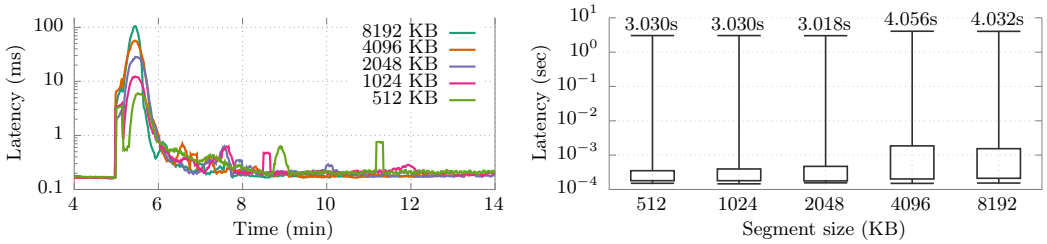

Fig. 38. Average restore latency during instant restore: time series (left) and distribution of individual values (right).

archiving process to reach the `restore_begin` log record; since the experiment leaves the archiving process running eagerly in the background, this wait is about 3 seconds only.

The results observed for this experiment show a clear trade-off between segment size and restore efficiency. Larger segments allow for higher bandwidth in the background single-pass restore process, which provides shorter total restore time. Smaller segments, on the other hand, reduce the restore latency as perceived by individual transactions in the beginning phase. To eliminate this trade-off and provide the best of both worlds, an adaptive technique can be used, in which the single-pass background service attempts to restore multiple adjacent segments whenever it encounters them. With this technique, small segments can be used, so that low latency is observed in the beginning phase. Then, during the end phase, background restoration performs large reads and writes of multiple segments, as if a much larger segment size would be used.

Results for this adaptive technique are shown in Figure 39. It uses segments of 1 MB, which can be coalesced into large units of up to 8 MB by the background restore process. The top chart shows the restore pattern, which clearly shows that overall restore lasts about as long (3 minutes) as for the largest segment in the previous experiment (8 MB) while the beginning phase has a dense pattern like the one observed earlier for the 1-MB segment size. The plots for restore bandwidth (bottom left) and latency (bottom right) also show that the advantages of small and large segments are combined.

3.5.3 Transaction throughput

The next experiment fixes the segment size at 1 MB with the adaptive technique and varies the buffer pool size. The goal is to evaluate the impact of the media failure and concurrent restore actions on running transactions. With a sufficiently large buffer pool, a media failure should go completely unnoticed, since all page accesses are hits, which do not trigger segment restore. In that case, the background restore process will perform single-pass restore, using the largest possible unit thanks to the adaptive technique described above. Therefore, this configuration should yield the shortest total recovery time. For smaller buffer pools, instant restore should cause a significant dip in transaction throughput, but it should gradually raise back up as segments of the working set are restored.

The buffer pool sizes considered here vary from 2.5 GB to 12.5 GB in increments of 2.5 GB. Figure 40 shows the results. In each chart, the x-axis shows elapsed time, as in the previous charts, whereas the y-axis shows transaction throughput. The shaded area starting at the five-minute mark shows the time during which instant restore was active. As the top-left chart shows, the smaller buffer pool incurs the longer restore time, since the high miss ratio causes almost all segments to be restored on demand. However, after about two minutes only, the pre-failure throughput is reestablished, thus demonstrating the main benefit of instant restore. With the remaining buffer sizes, the pattern is similar, but with higher transaction throughput. Note, for example, that with the 7.5-GB buffer pool,

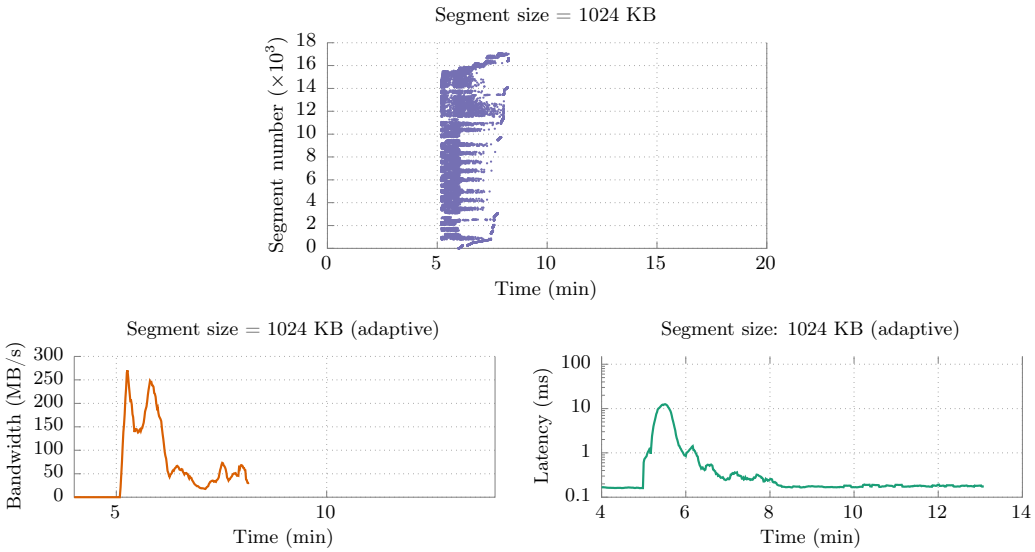

Fig. 39. Restore pattern, bandwidth, and latency with adaptive technique

restore finishes before the pre-failure throughput is reestablished; this is expected because restore uses frames of the buffer pool, and thus a higher miss rate is observed until all restored pages that are not in the working set are evicted.

With the buffer pool size of 12.5 GB, which is larger than the working set, the dip in throughput is very small, incurred mostly by the wait for the log archiver. This wait is only necessary because the buffer pool was probably not warm enough after the five-minute execution, and thus a few misses still occur. Unfortunately, the prototype system used here does not support a way to manually warm-up the buffer pool or to detect when it is fully warmed-up. Nevertheless, the results clearly demonstrate the effect of larger buffer pool sizes on instant restore—as it gets larger than the working set, the media failure goes practically unnoticed.

3.5.4 Log archiving overhead

The next experiments analyze the overhead of sorting the recovery log to produce log archive partitions. The sort logic consumes CPU cycles and memory bandwidth that could potentially be used for transaction execution. Furthermore, it utilizes I/O bandwidth to read the recovery log on one end and write to the log archive on the other. The latter is not a big concern, since transaction throughput does not depend on log archive bandwidth. The former, however, can potentially disturb transaction throughput if the consumed read bandwidth "steals away" from the write bandwidth required to sustain a high commit rate. These concerns can usually be mitigated if the log is cached in main memory and the log is archived eagerly, constantly, and at a steady pace, and also if the log device can support parallel reads and writes robustly—e.g., with modern SSDs.

Rather than evaluating the factors mentioned above individually using micro-benchmarks, this section uses a more pragmatical approach. A TPC-C benchmark with the same configuration as the earlier experiments of this section is executed for ten minutes under two different configurations: the first one simply copies and compresses the log to the archive device using the Linux `rsync` tool; the second one performs log archiving with sorting as described earlier. Both configurations run

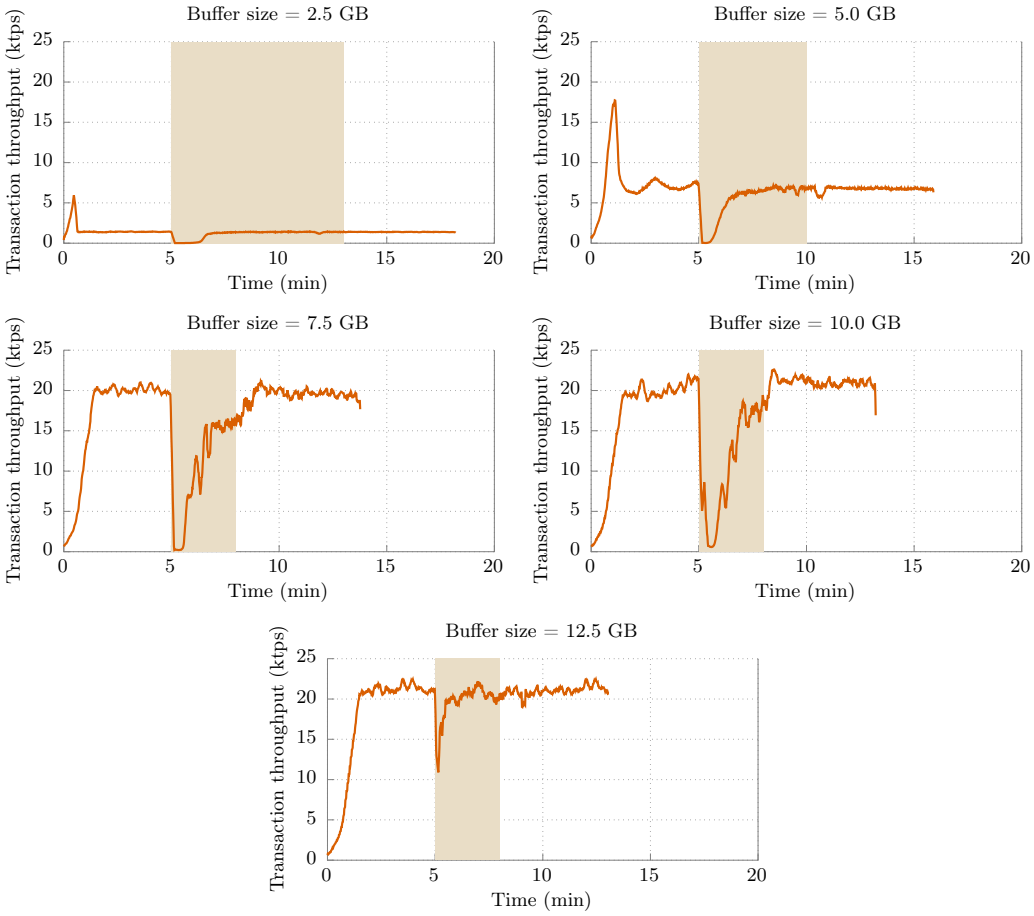

Fig. 40. Transaction throughput observed during instant restore with different buffer pool sizes

constantly in the background, and the experiments compare their transaction throughput and CPU utilization.

The first result, with both recovery log and log archive on SSD devices, is shown in Figure 41. On the left side, the distribution of transaction throughput values observed on each second of the experiment is plotted using a "box" plot of quartile values. The box contains values of the second and fourth quartiles, i.e., 50% of the observations, while the lines extending above and below the box reach to the maximum and minimum values, respectively. The red line in the middle is the median value. The same plot is used on the right side to show average CPU utilization on each second of the experiment.

As the results show, the "Sort" and "Copy" log archiving variants differ by a very small amount. With copying, the median transaction throughput lies slightly above 29 ktps; with sorting, it is approximately 28.75 ktps—a difference of less than 1%. With sorting, the maximum throughput is actually higher, but this is not a statistically significant result. As for CPU utilization, the first observation is that transaction processing—at least with this prototype implementation—never

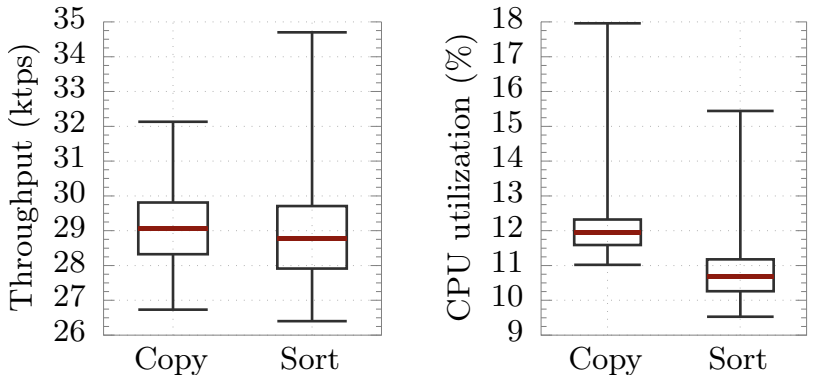

Fig. 41. Distribution of transaction throughput and CPU utilization values for two log archiving settings: copying vs. sorting—recovery log and log archive on SSD.

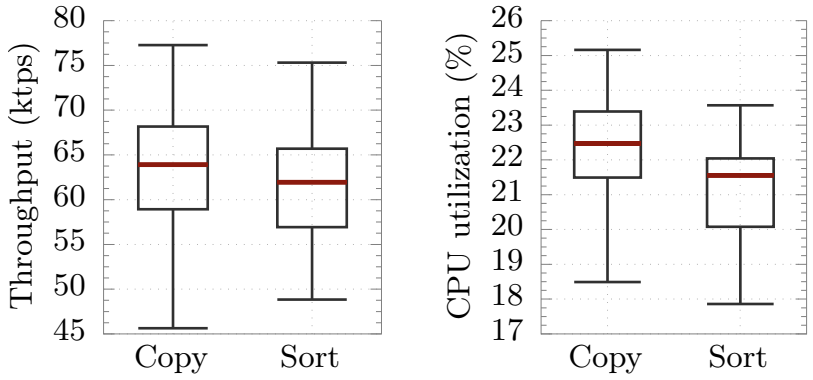

Fig. 42. Distribution of transaction throughput and CPU utilization values for two log archiving settings: copying vs. sorting—recovery log in DRAM and log archive on SSD.

consumes more than 20% of CPU cycles, despite the lack of logical (two-phase-locking) contention with a static thread-to-warehouse assignment in the TPC-C benchmark. In that case, the CPU cycles consumed by sorting are hardly a concern, especially considering that the compression applied in the copying variant also consumes CPU cycles. In fact, the results show that copying consumes about 10% more CPU cycles; some of that difference is likely due to the higher transaction throughput, but it shows that sorting does not incur a noticeable CPU overhead.

For the second experiment, the recovery log is placed in DRAM to maximize transaction throughput. The results are very similar to the previous experiments, except that transaction throughput is more than double than what is delivered with the log on SSD.

The conclusion is that log archiving with run generation using replacement selection, as implemented in this prototype, does not incur a significant overhead—i.e., less than 1%—in comparison with log archiving with simple file-system copying and compression.

4 PROPAGATION STRATEGIES

The recent advent of in-memory database systems has brought substantial improvements to transaction processing performance, by eliminating major architectural bottlenecks and better exploiting multi-core CPUs. However, the "in-memory" qualifier can be misleading, because practical, ACID-compliant systems must rely on and make effective use of persistent storage. The two major reasons for that are discussed below.

First, as argued in detail in Section 2.3, transaction durability requires persistence; thus, even if all data fits in main memory, the effects of committed transactions must be reflected on non-volatile storage. With a no-force policy, this is done with logging, but updates must still be propagated asynchronously in order to bound recovery time and free up log space. Most in-memory database systems make use of direct checkpoints for that [KN11, MWMS14, RDAT16a]. These checkpoints produce persistent files that are, for recovery purposes, essentially equivalent to a materialized database [HR83]. As such, they have much in common with traditional ARIES-based designs [MHL+92]. Therefore, the techniques of update propagation in a traditional ARIES setting can still be reused in a memory-rich environment, perhaps even with better results.

Second, the vast majority of workloads in practice have a working set that is much smaller than the total database size. Thus, it makes economical sense to offload portions of the dataset that are accessed less frequently into lower levels of the storage hierarchy, as suggested by the "five-minute rule" [GP87]. This is particularly relevant with the advent of fast SSD storage and emerging non-volatile memory, which bring the access latency of the lower levels much closer to that of main memory.

The two reasons given above translate into two crucial components of a transaction-processing system: first, it must employ an effective *propagation* strategy to bound recovery time and recycle log space; second, it must employ an effective *eviction* strategy to offload cold data into secondary storage. This chapter investigates how an abundance of main memory changes the assumptions that guided the design of these two components in traditional database system implementations. The focus here is on page-based techniques that work with a traditional buffer pool, preferably optimized for large main memory [GVK+14].

This chapter is organized as follows. Section 4.1 briefly reviews eviction strategies, discussing some related work and highlighting important implementation concerns that are crucial for high performance with memory-resident working sets. Section 4.2 discusses the design of an efficient *page cleaner*, which is the component responsible for asynchronous propagation using a buffer pool. Then, Section 4.3 follows up on that with an empirical analysis of various page cleaning strategies, evaluating their effectiveness in bounding recovery time, enabling efficient page eviction, and sustaining high transaction throughput. Finally, Section 4.4 presents and evaluates novel log-based techniques to perform page cleaning.

4.1 Page eviction

One of the key concerns of database buffer management is *page replacement* [EH84]. The term "replacement" implies that a buffer pool frame is only freed on demand, to make room for a page being fixed by a transaction. However, freeing up frames in the buffer pool could also be done more proactively, in order to always maintain a small percentage of buffer pool frames in the free state. This can be beneficial for workloads with a high insertion rate, in which case latency spikes incurred by page replacement can potentially be eliminated. For these reasons, this chapter employs the more general term *page eviction* to denote the selection of buffer pool frames to be freed.

Strategies for page eviction have been developed since the early days of database systems [EH84] and have continued to be a topic of active research since then [OOW93, MM03, BM04, JCZ05, YMUY10, ELL14, MAZ+16]. This section is not concerned with the discussion and evaluation of these many strategies; rather, it highlights some of the key concerns for page eviction in memory-abundant scenarios, where it is crucial to eliminate performance bottlenecks.

Page eviction strategies must collect information about page references during runtime, in order to maintain statistics about what pages are good candidates for eviction. Typical implementations involve some type of callback built into the buffer pool *fix* method; it simply lets the eviction component know that a certain page has been referenced. For brevity, this callback is called *ref* here.

A call to *ref* includes the page ID and optionally additional metadata such as the latch mode (shared or exclusive). A memory-optimized system can potentially produce millions of *ref* calls per second from multiple threads. Therefore, it is absolutely crucial for performance that this function does not incur any synchronization overhead, performing only local, preferably seldom, updates to eviction data structures. This requirement precludes the use of strategies that require updating a global linked list, such as LRU or its modern incarnations (e.g., LRU-k [OOW93] or ARC [MM03]).

To avoid the overhead of global data structure updates on every *ref* call, CLOCK-based approaches are well suited (e.g., CAR [BM04], ClockPro [JCZ05], or NbGCLOCK [YMUY10]). These approaches only update a bit or counter stored in each frame control block, and thus eliminate synchronization overheads and minimize cache misses. Generalized strategies that employ a clock counter rather than a bit usually produce improved hit ratio, but they might incur additional *ref* overhead because a shared-memory location (the counter) is updated on every call, which increases cache coherence traffic. A single bit, on the other hand, only requires an update if the bit is currently set to zero.

These concerns point to a fundamental rule for designing page eviction strategies for high-performance transaction processing: low-overhead page references have higher priority over high hit ratios. This is especially relevant if the latency of a page miss (i.e., the penalty of a bad eviction decision) is relatively low, as is the case with modern SSD storage and emerging non-volatile memory. Therefore, future hardware architectures might favor simple, low-overhead strategies with moderate hit ratio (e.g., CLOCK or even RANDOM [EH84]) over sophisticated strategies with higher hit ratio.

An alternative to reduce the overhead of *ref* calls is to not invoke the method on every reference, relying instead on some sort of sampling technique. The anti-caching approach [DPT+13], for instance, maintains an LRU linked list of tuples that is updated only by a small, randomly-selected subset of transactions. Other approaches rely on an offline analysis of a tuple-reference log [SA13, ELL14], in which case the *ref* call must append a new entry this log. To reduce the overhead of logging, sampling is also employed. Despite these techniques being tuple-oriented, the same principles discussed here for page-oriented eviction apply. In essence, these techniques also aim to fulfill the fundamental rule discussed above—i.e., to reduce reference overhead at the expense of hit ratio. Since the goal here is not to evaluate different eviction strategies, the remainder of this chapter assumes a traditional CLOCK-based strategy on a buffer pool, without sampling.

Lastly, the role of pointer swizzling in the buffer pool, as proposed by Graefe et al. [GVK+14] may directly influence the design of page eviction strategies. Pointer swizzling eliminates page ID lookups from the critical path of transactions referencing a page, which accounts for almost all the overhead measured in the study by Harizopoulos et al. [HAMS08]. Pointer swizzling adds another dimension to the life cycle of buffer pool frames, namely the *swizzled/unswizzled* state, which could be exploited to effectively identify eviction candidates without any *ref* overhead on swizzled pages. This chapter assumes a traditional buffer pool without pointer swizzling, but the concepts discussed here can be easily adapted (or even improved) in a design supporting swizzling.

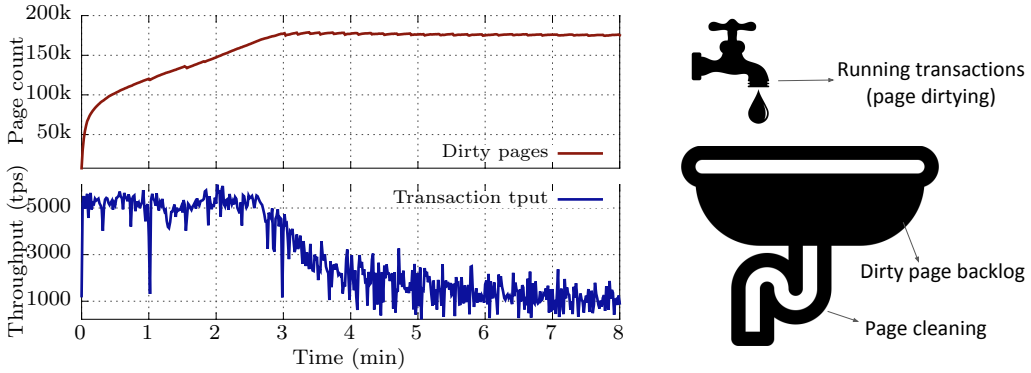

Fig. 43. Dirty page backlog and its implication on system performance

4.2 Page cleaner implementation

The page cleaner component of a database system buffer pool is responsible for asynchronously propagating updates into the materialized database, following a no-force approach [HR83]. In a scenario where transaction throughput is limited by I/O bandwidth—which was typical when OLTP systems first arrived—such propagation usually happens on demand, as a consequence of evicting a page. However, as the cost of main memory decreases and larger portions of an application's working set fit into the buffer pool, running transactions are less likely to depend on page I/O to make progress. In this scenario, update propagation plays a more independent and proactive role, where the main goal is to control the amount of cached dirty data. This is crucial to maintain high performance as well as to reduce recovery time in case of a system failure.

4.2.1 Motivation

The problem addressed in this section can be characterized by a race between user transactions that modify pages and mark them dirty and system actions that clean these pages. If the *cleaning speed*, i.e., the number of pages being cleaned per second, does not match the *dirtying speed* of the workload, the accumulated backlog may negatively impact system performance. This backlog can be measured across two dimensions: number of dirty pages and accumulated log volume. In the latter case, the issue appears when the log volume required for redo recovery—here referred to as the *redo length*—fills up the entire log device, so that transactions cannot make progress until some log space is freed. Since the redo length is determined by the set of dirty pages in the buffer pool (more precisely their minimum recovery LSN value), this means that an inefficiency in page cleaning can lead to a complete halt of read-write transactions. In the former case, if the number of dirty pages grows until it fills up the entire buffer pool, the system eventually slows down as transactions must wait for page eviction, despite their working set fitting into main memory. This can happen, for instance, in the TPC-C workload, whose working set consists mainly of warehouse and customer data as well as currently active orders. If page cleaning is inefficient, dirty pages containing finished orders will linger in the buffer pool, until no clean frames are available for inserting new orders and the system slows down, becoming I/O-constrained even though there is abundant main memory to hold the working set.

Figure 43 presents the problem graphically in two ways. On the right-hand side, the problem is illustrated as an analogy of a sink full of water—running transactions that make clean pages dirty

are like a faucet filling up the sink, while page cleaning corresponds to the drain. If the drain is not large enough, water will accumulate in the sink, which in our case corresponds to the backlog discussed above. Eventually, the sink fills up and the only way to avoid an overflow is to close the faucet, i.e., the transaction throughput must be reduced. On the left-hand side, the problem is shown in a real experiment which plots both the number of dirty pages in the buffer pool as well as the transaction throughput over time. On the top graph, the number of dirty pages grows until it reaches the buffer pool size of 180,000 pages. At that point, which occurs at minute 3 of the experiment, the transaction throughput drops substantially, from 5,000 to about 1,000 transactions per second. Such drop in throughput is a direct consequence of the page cleaner not being able keep up with the running transactions.

Techniques for efficient page cleaning addressing the problems mentioned above have not been addressed by current database research. Related work on buffer management techniques usually just assumes that pages are written in the background [JPH+09, GVK+14, YMUY10], without providing further details or discussing effective ways to reduce recovery backlogs. The remainder of this section aims to fulfill this research gap by discussing implementation techniques for a modern page cleaner.

4.2.2 Asynchronous cleaning without page latches

Page cleaning under high transaction volume has two goals that could be conflicting under a naive implementation: on one hand, dirty pages should be written out quite aggressively to minimize the recovery backlog discussed earlier; on the other hand, page cleaning should interfere as little as possible with running transactions, in order to sustain high transaction throughput. To minimize such interference, the page cleaning thread should not hold a latch (even if in shared mode) throughout the entire duration of a synchronous write-through operation.

A simple technique to minimize the period during which a page is latched for cleaning is to reserve a small fraction of main memory space for a *cleaner buffer*. In order to write a page, the cleaner then acquires a shared latch, copies the page's contents into the buffer, and immediately releases the latch. The problem with this approach is that after the page is written, the page cleaner cannot simply mark the corresponding frame as clean in the buffer pool, because further updates might have been applied to the page in the meantime. Furthermore, it must also update the recovery LSN value correctly, regardless of whether further updates have happened in the meantime or not.

To address these challenges, buffer pool control blocks must keep track of more information than a simple "dirty" Boolean flag and a single recovery LSN value. Before describing how asynchronous cleaning without page latches can be performed correctly, the following paragraph presents an example situation that can result in incorrect recovery with a Boolean flag.

Figure 44 presents a similar situation to that of Figure 6 in Section 3.1.2, which discussed how to determine dirty pages during log analysis without false negatives. In this new example, page A is copied into the cleaner buffer with contents A_1 on LSN 110, after which it is updated in the buffer pool to new contents A_2 on LSN 120. Then, on LSN 130, page A is flushed into the database with contents A_1, and it is *incorrectly* marked as clean in the buffer pool. This situation can result in the loss of update $A_1 \rightarrow A_2$ if there is a system failure, because a checkpoint—performed on LSN 140—will not include A in its list of dirty pages. This example clearly demonstrates that the page LSN of the flushed copy and the current page LSN in the buffer pool must be compared before a page can be considered clean.

Before presenting a solution for correct asynchronous page cleaning, another problem must be considered, namely the determination of the new *recovery LSN* value after cleaning a page. As

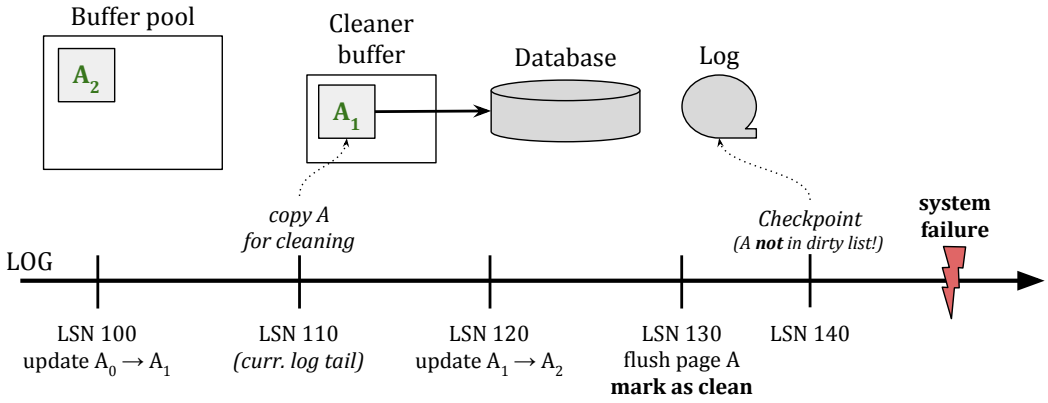

Fig. 44. Lost updates as a result of incorrect page cleaning

discussed earlier, the recovery LSN refers to the first update to a page since it was last cleaned; it determines the first log record to be replayed on that page during restart recovery. Even if the recovery LSN is not required for restart recovery, as is the case with instant restart, the buffer pool must still keep track of it to determine the restart low-water mark that guides log space recycling. For this reason, the recovery LSN value in the corresponding control block must be updated every time a page is cleaned—regardless of whether the page actually becomes clean or not.

In the example of Figure 44, page A should not be marked clean after it is flushed, because a new update has been made on LSN 120. Nevertheless, the recovery LSN must be updated to 120, as that is now the LSN of the first log record that will be replayed on A during redo recovery after the system failure.

Given these two problems to be addressed—correctly determining the dirty state of pages and keeping track of their recovery LSNs—the following asynchronous page cleaning algorithm is proposed.

Since LSN values must be compared to determine whether a page in the buffer pool is actually clean or dirty, the algorithm eliminates the "dirty" Boolean flag, relying instead on a control-block field called `persistedLSN`. This field contains the page LSN of the page on the materialized database. In the example of Figure 44, it would be 100 after the page flush is completed. Using this field, a page is considered clean if and only if its page LSN equals its `persistedLSN`.

To update the persisted LSN as well as the recovery LSN correctly with asynchronous page cleaning, two additional fields are maintained in each control block: `nextPersistedLSN` and `nextRecoveryLSN`. The former is set when the page is copied for cleaning by saving the current page LSN, at which point the latter is also set to a reserved `null` value that sorts lower than any other LSN value (typically zero). These two fields are updated while holding the shared latch acquired for copying the page.

When a page is updated, i.e., its page LSN is incremented, two conditional updates are performed on control-block fields: first, if this is the first update since the page was last cleaned (i.e., if `persistedLSN` ≥ `recoveryLSN`), then `recoveryLSN` is set to the current page LSN; second, if this is the first update since the page was last copied for cleaning (i.e., if `nextPersistedLSN` ≥ `nextRecoveryLSN`), then `nextRecoveryLSN` is set to the current page LSN.

Finally, when a page is flushed by the cleaner, `recoveryLSN` is set to `nextRecoveryLSN` and `persistedLSN` is set to `nextPersistedLSN`. These updates must be performed atomically, which can be guaranteed by acquiring a shared latch on the page.

The tracking of these LSN values for management of dirty state is presented as pseudo-code functions in Algorithm 5. The *Initialize* function initializes a control block *cb* with a given page LSN *plsn*; it is invoked when allocating a frame for a new page or when fetching a page from the database. As the comment on the right side indicates, the caller must hold an exclusive latch on the frame before calling this function; the remaining functions are also annotated accordingly. While latching could be performed internally, or with a different concurrency control mechanism such as atomic instructions or transactional memory, these potential optimizations are ignored here.

The next function, *IsDirty*, shows how to inquire the dirty state of the page as discussed above, replacing the traditional Boolean flag. The function *UpdatePageLSN* is invoked whenever an update is performed on the page. The algorithm assumes that the page LSN value is replicated in the control block structure for simplicity, but in practice it could be stored exclusively on the page. Besides setting the new page LSN, *UpdatePageLSN* also sets the recovery LSN values accordingly, if the present update is the first since the page was last cleaned or copied for cleaning.

The last two functions, *MarkPersistedLSN* and *NotifyWrite*, are invoked by the page cleaner. The former is called when a page is copied into the cleaner buffer, while the thread still holds a shared latch on the frame. It saves the "next" values that will be set in the latter function, which is called by the cleaner after the write is completed successfully. Note that these two functions only require shared latches, even though they write into control-block variables. This suffices because the functions never race with each other, since the cleaner—the only component that invokes these functions—is executed with a single thread. Furthermore, these functions do not conflict with *UpdatePageLSN*, because the latter requires an exclusive latch. If multiple cleaner threads are employed, the space of page identifiers should be partitioned among them (e.g., by table-space or by device) so that they never interfere with each other.

One observation suggested by this cleaning mechanism with LSN bookkeeping is that page cleaner actions can be seen as system transactions, because they must be perceived as atomic and the visibility of their effects (i.e., their durability) is vital for correct recovery. As such, cleaner actions can be implemented and described more explicitly as system transactions. The mechanism presented here is analogous to a shadow-based commit protocol—the saved values of the "next" fields in the control block are basically shadow versions of their permanent counterparts, which are installed atomically when a page write is completed, or, under this new interpretation, *committed*. Given this perspective, page cleaning can also be implemented with regular system transactions that make use of write-ahead logging. For example, when a page is copied for cleaning, that action can be logged along with the page LSN in a system transaction; the same holds for the final completion of the write operation. If a system failure occurs, log analysis must reconstruct, along with the list of dirty pages, a list of such "in-flight" writes. This would allow the restart process to correctly determine the list of dirty pages, solving the lost-update problem illustrated in Figure 44. It is also worth pointing out that this alternative technique would be somewhat similar to strategies employed for asynchronous incremental backups [MN93].

Using the control-block infrastructure presented here, an asynchronous page cleaning service can be implemented without violating ACID properties. The next section discusses how to implement such a service.

Algorithm 5 Functions used to manage page cleaning in buffer pool control blocks

```
 1: procedure Initialize(cb, plsn)                    ▷ (caller must hold exclusive latch)
 2:     cb.pageLSN ← plsn
 3:     cb.persistedLSN ← plsn
 4:     cb.recoveryLSN ← null
 5:     cb.nextPersistedLSN ← null
 6:     cb.nextRecoveryLSN ← null
 7: end procedure

 8: procedure IsDirty(cb)                             ▷ (caller must hold shared latch)
 9:     return cb.pageLSN > cb.persistedLSN
10: end procedure

11: procedure UpdatePageLSN(cb, plsn)                 ▷ (caller must hold exclusive latch)
12:     cb.pageLSN ← plsn
13:     if cb.persistedLSN ≥ cb.recoveryLSN then
14:         cb.recoveryLSN ← plsn
15:     end if
16:     if cb.nextPersistedLSN ≥ cb.nextRecoveryLSN then
17:         cb.nextRecoveryLSN ← plsn
18:     end if
19: end procedure

20: procedure MarkPersistedLSN(cb)                    ▷ (caller must hold shared latch)
21:     cb.nextPersistedLSN ← cb.pageLSN
22:     cb.nextRecoveryLSN ← null
23: end procedure

24: procedure NotifyWrite(cb)                         ▷ (caller must hold shared latch)
25:     cb.persistedLSN ← cb.nextPersistedLSN
26:     cb.recoveryLSN ← cb.nextRecoveryLSN
27: end procedure
```

4.2.3 Page cleaner service

The previous section introduced a mechanism to clean individual pages asynchronously while guaranteeing transaction correctness. This section discusses how to implement a page cleaner service that determines which pages to clean and how the write of such pages should be scheduled. This is crucial in order to effectively fulfill the goals laid out in the beginning of this chapter: bound recovery time in case of a system failure and allow continuous log space recycling, all while keeping up with the high transaction throughput produced with memory-resident workloads.

The page cleaner is executed in *rounds*, which can be scheduled to run at fixed time intervals, triggered by system activities such as page eviction, or simply run constantly in the background without pauses. Each round has three major steps: first, the buffer pool is scanned to collect a list of *candidate* frames; then, some candidates are *copied* into the cleaner buffer until it fills up; finally, the buffered pages are *flushed* to the database. If the list of candidates is larger than the buffer, the last two steps can be repeated until all candidates are consumed.

A naive implementation of the page cleaner—as implemented in the original Shore-MT [JPH+09], for instance—simply scans the whole buffer pool and immediately flushes all dirty pages it finds. As the next section shows, such implementation is quite ineffective, for two main reasons: first, it performs single-page writes only, thus not being able to exploit larger writes for higher cleaner bandwidth; second, not all dirty pages are worth cleaning at a particular point in time, because some of them are likely to not contribute significantly to reducing the recovery backlog. To see why the second point is important, consider a page that is updated very frequently, but only for a small period of time—e.g., the last page of an append-only table. Cleaning this page within this period basically incurs a wasted I/O, because the page will very soon become dirty again and its recovery LSN is most likely only increased by an insignificant fraction of the total redo low-water mark. Given these reasons, the first phase of a cleaner round—candidate selection—is crucial to achieve effective page cleaning.

Candidate selection is subject to a *policy*, which is a comparison function used to rank the dirty frames encountered from most to least important. Specific policies and their empirical evaluation will be presented in Section 4.2.4 below; for now, it can be simply considered as an arbitrary comparison function. The policy is used to construct a priority queue of frames ranking highest among all inspected dirty frames. The number of candidate frames, and thus the size of the priority queue is fixed as a fraction of the total buffer pool size (e.g., 1%).

In the second phase, the ranking established by the comparison function can be ignored, i.e., the priority queue is interpreted as a simple array of frame descriptors. This array then is sorted by the page location on persistent storage, the goal of which is to build *clusters* of adjacent pages that can be flushed with a single write. Then, the frames in each cluster are latched in shared mode and the page's contents are copied into the cleaner buffer. When taking a copy, the *MarkPersisted* function described earlier must be invoked on the frame's control block. The latch can be acquired conditionally in order to reduce contention on hot-spot pages, essentially providing a best-effort approach. In this case, the cleaner must give up on a contented page that could not be latched conditionally, which might break the contiguousness of a cluster and must thus be handled accordingly.

Finally, when the cleaner buffer is filled, a write operation is invoked for each cluster. A key challenge here is to exploit I/O parallelism in order to maximize the cleaner throughput. Two aspects make this non-trivial: first, the cleaner should not spawn multiple threads to perform page writes in parallel, because the scheduler overhead can easily interfere with transaction processing; second, to guarantee ACID properties, the persisted and recovery LSNs can only be updated (by the *NotifyWrite* function introduced earlier) after the successful completion of a write, which precludes the use of asynchronous I/O libraries (e.g., `io_submit()` in Linux [Tar12]).

Given the restrictions above, a simple and effective way to achieve I/O parallelism is to rely on the I/O schedulers of lower levels of the system—e.g., operating system, RAID controller, flash translation layer, etc. To that end, non-blocking `write()` system calls can be issued at any point after a frame is copied into the cleaner buffer. Then, in the write stage of the cleaner round, after one write has been issued for each cluster, the `fsync()` system call is issued to make sure that all pending writes are forced to persistent storage. After this step, the cleaner iterates over the flushed frames and updates the LSNs in their control blocks with the *NotifyWrite* function. If more candidates were collected in the first phase than fit in the cleaner buffer, the remaining candidates are copied and flushed until all have been processed, after which the cleaner round is finally finished.

The pseudo-code for a cleaner round and its phases is presented in Algorithm 6. It shows a simplified version of the algorithm described above for brevity, but the main phases and elements

are present. Note the use of a *cleanLSN* field that is saved on page-flush log records—its purpose was explained in Section 3.1.2.

Algorithm 6 Function that executes one round of the page cleaner service

```
1:  procedure CleanerRound
2:      pq ← ∅
3:      cmp ← GetCleanerPolicy()
4:      cleanerBuffer ← AllocateBuffer(NumCandidates)
5:      for cb in BufferPool.ControlBlocks do
6:          if IsDirty(cb) then
7:              pq.insert(cb, cmp)
8:              if pq.size() > NumCandidates then
9:                  pq.removeTop()
10:             end if
11:         end if
12:     end for
13:     clusters ← SortAndAggregateByPageID(pq)
14:     cleanLSN ← CurrentLogTail()
15:     pos ← 0
16:     for c in clusters do
17:         for cb in c do
18:             CopyFrame(cb, cleanerBuffer[pos])
19:             MarkPersistedLSN(cb)
20:         end for
21:         pos ← pos + 1
22:         AsyncWrite(c, writeBuffer, pos)
23:     end for
24:     fsync()
25:     for c in clusters do
26:         for cb in c do
27:             NotifyWrite(cb)
28:             LogPageFlush(cb.pageID, cleanLSN)
29:         end for
30:     end for
31: end procedure
```

The next section discusses cleaner policies and presents an empirical evaluation of their effectiveness in controlling the recovery backlog.

4.2.4 Page cleaner policies

A page cleaner policy is defined as a sort order applied to candidate dirty pages when performing a page cleaner round. This is implemented using a priority queue (*pq*) with a parametrized comparison function (*cmp*) in Algorithm 6. The main efficiency measure of the page cleaner is its write bandwidth, i.e., how many pages it can write per second (or how large the "drain" is in the sink analogy of Figure 43). However, as discussed earlier, optimizing for write bandwidth does not necessarily minimize the recovery backlog, because the redo length of the log must also be reduced. If the cleaner policy

in use neglects the redo length, a situation similar to that of Figure 43 may happen when the log device is full. Therefore, the goal of page cleaning policies is to reduce both the number of dirty pages in the buffer pool as well as the redo length. This section proposes and empirically evaluates policies based on this goal.

Cleaner policies reuse the fields in a frame's control block to provide the desired ranking. One straight-forward policy that is often used in practice is to pick the frames with the *oldest recovery LSN*. The rationale here is to minimize the redo length by making sure that the minimum recovery LSN is increased with every round of the page cleaner. In other words, this policy takes considers how long a page has remained dirty in the buffer pool.

One potential downside of prioritizing pages with oldest recovery LSN is that it does not directly aim to minimize the number of dirty frames in the buffer pool. As mentioned earlier, for workloads like TPC-C, the page cleaner should prioritize pages containing finished customer orders, so that they can quickly be evicted and make room for new orders. Warehouse and customer data, on the other hand, are updated randomly, which means they should be allowed to linger in the buffer pool longer than finished orders. While proposing such workload-tailored policies is not the goal of this chapter, it is important to identify such patterns and match them with policies that handle them effectively. In this particular case, a cleaner policy that prioritizes recency or density (references per unit of time) of page updates might perform better.

Cleaner policies are very similar to page eviction strategies [EH84], in which they use simple reference statistics to determine which pages should be selected. The policies employed may take different criterion into consideration, such as age, recency, frequency, density, etc. Despite the similarities, a memory-abundant scenario benefits from keeping these two concerns–eviction and cleaning—separated, since controlling recovery backlog becomes a more important issue. Furthermore, page cleaner policies only consider dirty pages, meaning that only statistics about page updates—and not reads—are relevant. Eviction, on the other hand, considers page references in general, regardless of their type.

This work also considers two straight-forward, but naive page cleaner implementations: *eager cleaning* and *eviction-oriented cleaning*. The former simply scans the buffer pool and flushes every dirty page, similar to the one implemented in the original Shore-MT [JPH+09]. The latter is actually the absence of a page cleaner service: instead of using a background thread, pages are cleaned only by the eviction component, whenever the chosen eviction candidate happens to be dirty.

4.3 Evaluation of page cleaning strategies

This section evaluates different page cleaning strategies on the following criteria: transaction throughput, write bandwidth, number of dirty pages in the buffer pool, and redo length. The experiments below rely on the same environment and workload as the instant restore experiments in Section 3.4, but with a smaller database of 10 GB, stored on an SSD device. The TPC-C benchmark is executed for 25 minutes with 12 worker threads. The focus here is on a scenario in which the working set of the application barely fits in the buffer pool, which thus also has a size of 10 GB. The goal is to measure how well propagation strategies deal with main-memory sizes large enough to deliver high performance but not too large so that efficient propagation is not a concern. This should, in practice, be the most economically favorable setup.

Note that the TPC-C benchmark grows over time as new orders are inserted and processed, but the working set (warehouse and customer data) stays roughly the same size. Therefore, the setup considered here models the backlog situation with the sink analogy presented in Figure 43; an efficient page cleaner should be able to quickly get rid of pages containing finished orders and retain

only the working set in main memory. As the experiments below show, this is unfortunately hard to achieve with existing techniques.

The experiment results in this section will be presented in groups of four charts, as shown in Figure 45. All charts use time in the y-axis, with data points collected at the granularity of one second. The top-left chart plots transaction throughput (in thousands per second) on the left y-axis, while the right y-axis shows the average time it takes (in milliseconds on a log-scale) to evict a free frame in the buffer pool. As such, values greater than zero indicate that the buffer pool has become full. The top-right chart shows propagation efficiency across its two main measures: number of dirty pages against the buffer pool, plotted on the left y-axis, and the redo length (in GB), plotted against the right y-axis. The two bottom charts display I/O metrics collected for reads and writes on both the log and the database devices with the `iostat` tool. The bottom-left chart shows bandwidth in MB/s and the bottom-right one the number of I/O operations executed per second (IOPS).

4.3.1 Eviction-oriented page cleaning

The first propagation technique considered is a naive eviction-based approach, in which there is no dedicated page cleaner service. Instead, pages are written only when picked as victims by the eviction algorithm.

As the results in Figure 45 show, a maximum transaction throughput just below 30 ktps is quickly reached, but as soon as the buffer pool fills up, it collapses to under 5 ktps in average. This result is very surprising—it shows that random reads are very fast on the SSD considered here, providing very fast warm-up and delivering nearly the maximum throughput measured in the instant restart experiments of Section 3.2. The immediate reason for the throughput collapse is that, as the buffer pool becomes full, it takes a significant amount of time—2 ms in average—for a worker thread to evict a page and obtain a free frame. This delay is almost equivalent to the I/O latency of an HDD device. Going one level deeper, the ultimate cause for this problem is the propagation inefficiency—as the I/O metrics on the bottom charts show, dirty pages are being flushed at a rate of 6 MB/s only, despite the relatively high average IOPS values above 2,000.

The extremely low write bandwidth of the eviction-oriented cleaning is due to the inefficiency of synchronous single-page writes, which are required to guarantee ACID properties. To confirm that the results observed here are not due to a systemic error, the `fio` I/O benchmark tool was executed with similar configuration: single-page (8 KB) writes with 12 worker threads and `fsync` after every write. The results were slightly better than observed here—9 MB/s and 1,200 IOPS. The lower performance observed in the eviction experiment is likely due to the fact that, unlike in the `fio` experiments, TPC-C worker threads also spend time reading pages ans executing transactions. The TPC-C experiment also incurs additional propagation overheads than simply writing pages, such as the need to observe the WAL rule (i.e., make sure the log is durable until the page LSN of each page), the frame search using the CLOCK algorithm, and the latching overhead.

The chart on the top-right of Figure 45 shows that this strategy is very ineffective at maintaining a low dirty-page ratio in the buffer pool. Unsurprisingly, it is nearly at maximum level at all times after the buffer pool becomes full. Also unsurprisingly, the redo length grows continuously; this is because, as explained earlier, hot pages that are never picked by the eviction algorithm will linger in the buffer pool and never be flushed to the database. This means that recovery time is practically unbounded, which makes this approach unusable in practice; it is considered here solely to study propagation efficiency.

Lastly, to investigate the effect of the eviction strategy, the same experiment was executed with random eviction. The results are shown in Figure 46. As confirmed by the plot on the left side,

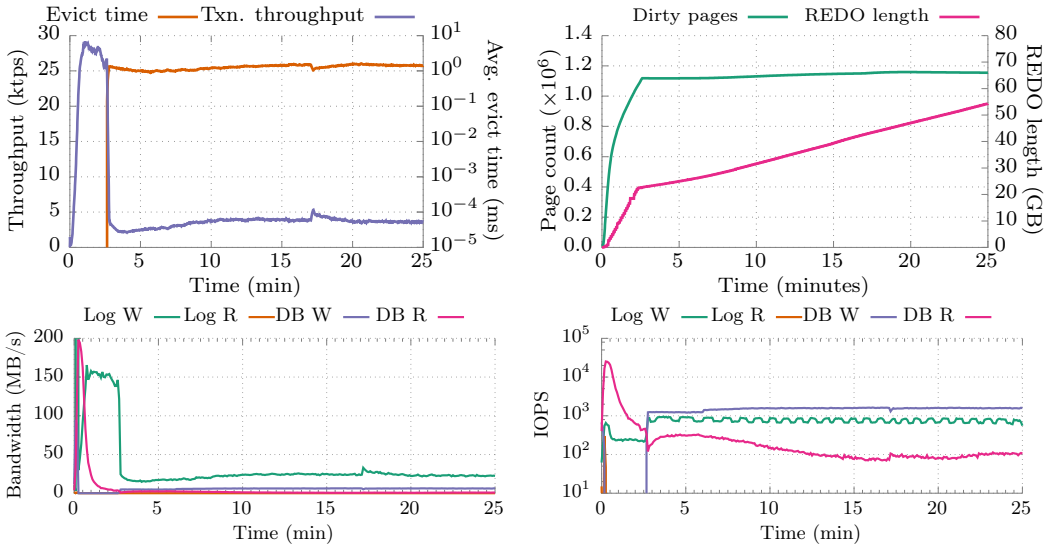

Fig. 45. Propagation efficiency and I/O metrics of eviction-oriented cleaning (CLOCK eviction)

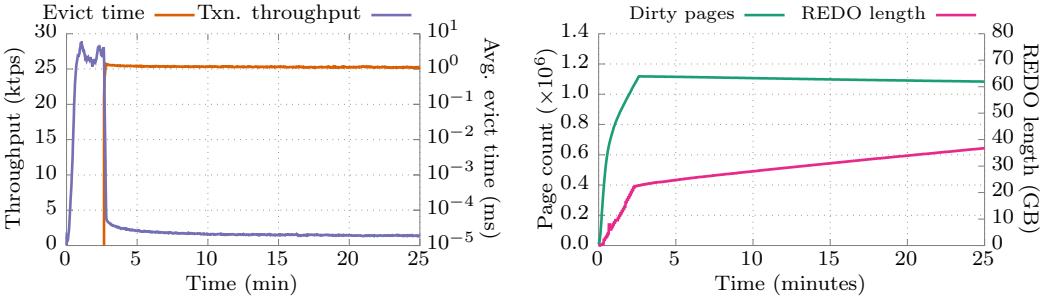

Fig. 46. Propagation efficiency of eviction-oriented cleaning (RANDOM eviction)

average eviction time is indeed lower, but only marginally so and actually barely noticeable in the graph. Transaction throughput, on the other hand, collapses even further than the CLOCK approach, to ~2 ktps.

The conclusion of this experiment is that eviction-oriented page cleaning is unrealistic for scenarios in which the working set fits in the buffer pool—not only due to the inability to bound recovery time, but also to the extremely low write performance.

4.3.2 Eager page cleaning (no policy)

The next experiment introduces a page cleaner service that runs constantly in the background. In this case, eviction never writes a dirty page—it simply keeps searching the buffer pool until it finds a clean eviction victim.

The eager page cleaner uses the same infrastructure as the three-phase asynchronous cleaning algorithm described earlier, but with slightly different behavior. First, candidate frames are collected and copied into the cleaner buffer. However, since there is no policy involved, any dirty page is immediately copied, without maintaining a priority queue. As soon as the buffer fills up, one

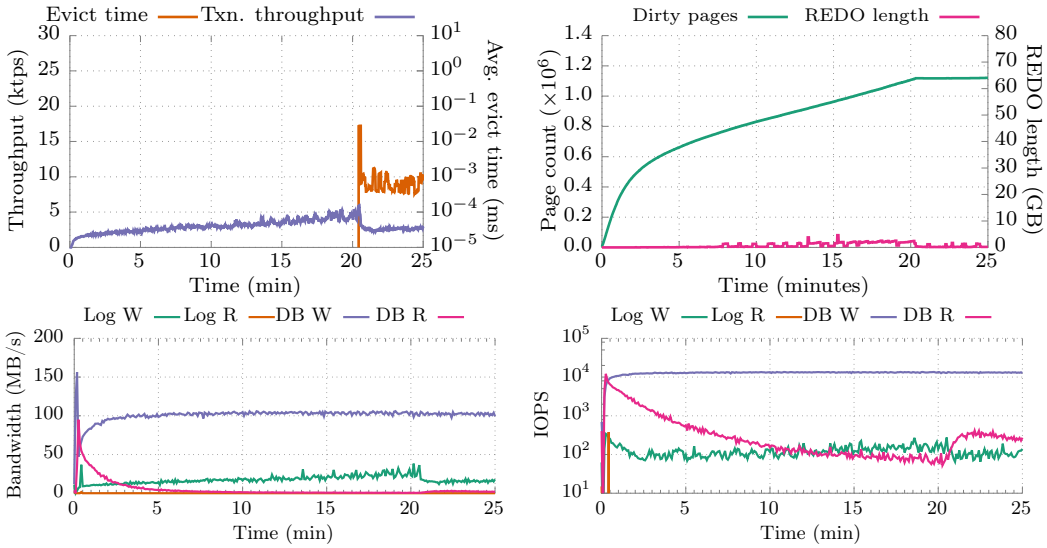

Fig. 47. Propagation efficiency and I/O metrics of eager page cleaning

asynchronous `write()` call is invoked for each page in the buffer. Then, the `fsync()` system call is invoked to guarantee that all writes are persisted, after which the control blocks of the flushed pages are updated to reflect the new persisted LSNs.

Results for this experiment are shown in Figure 47. The primary observation from these results is the significantly better write bandwidth, slightly under 100 MB/s. This is an immediate consequence of multiple asynchronous write calls followed by a single `fsync()` invocation, rather than writing each page synchronously. In this way, despite all writes being of single pages, multiple write requests can be buffered in the lower levels of the system (e.g., in the operating-system I/O scheduler) and propagated at once.

The improved cleaning speed results in higher transaction throughput, which peaks at ~5 ktps at the 20-minute mark, which is exactly when the buffer pool fills up. After that, throughput drops ~2.5 ktps and seems to recover back up to ~4 ktps. This improved, albeit more fluctuating, performance is also reflected in the average eviction time, which is slightly below 1 μs, or three orders of magnitude lower than with eviction-oriented cleaning.

On the negative side, the fact that it takes twenty minutes instead of one to fill up the buffer pool is a major disadvantage of this cleaning strategy. This is because the random page reads performed to warm up the buffer pool must now compete with write traffic from the page cleaner. This is especially visible in the IOPS measurements, where database write activity steadily delivers 10,000 IOPS in comparison with the 1,000 IOPS observed with eviction-oriented cleaning.

In terms of bounding recovery time, this cleaner strategy is very effective at controlling redo length, which remains within the 1–2 GB range throughout the experiment. However, it seems like page cleaning efficiency can be improved, since the number of dirty pages in the buffer pool is kept close to maximum after the buffer fills up. Note that this behavior is expected to scale with the database size—larger datasets would exhibit proportionally higher number of dirty pages.

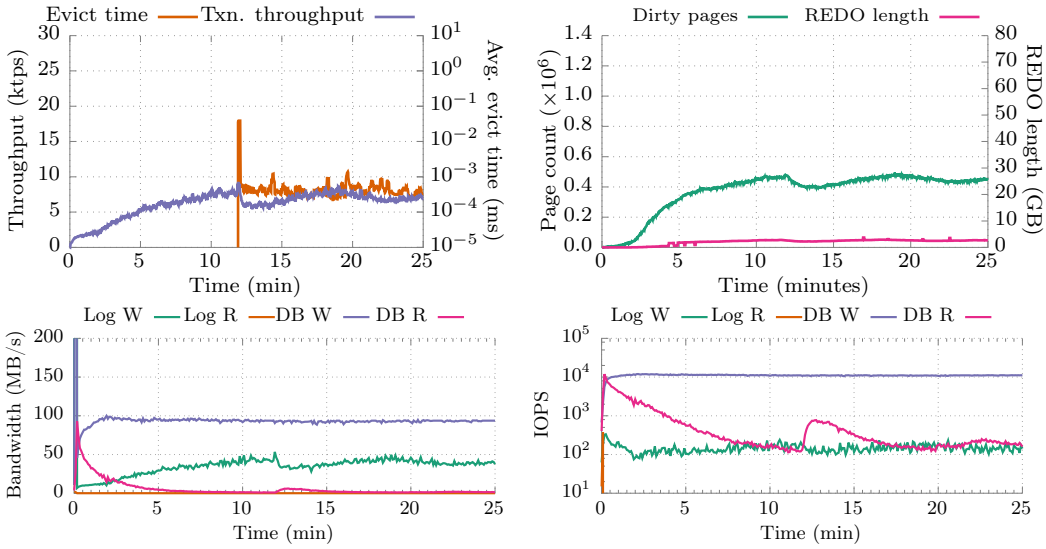

Fig. 48. Propagation efficiency and I/O metrics of page cleaning with oldest-LSN policy

4.3.3 *Policy-based cleaning*

The next experiments evaluate page cleaning with policies, i.e., with candidate collection in a priority queue, according to different comparison functions. Frames to be flushed are also grouped into clusters of neighboring page identifiers, which can be flushed with a single write. The goal of these more elaborate cleaning strategies is to deliver maximum write bandwidth, as in the eager strategy, but with reduced recovery backlog, i.e., less dirty pages and lower redo length. If a higher percentage of the frames in the buffer pool are kept in the clean state, it means that eviction would find victim frames much faster, thus directly improving transaction throughput as well.

In the first policy analyzed is, the frames with the lowest recovery LSN value are picked for cleaning; this policy is referred to as oldest-LSN. Each cleaner round selects 10,000 candidates, which is less than 1% of the buffer pool size. This is also chosen as the size of the cleaner buffer, so that all picked candidates are flushed with a single round.

The results are shown in Figure 48. In comparison with the eager strategy, this policy delivers higher transaction throughput, which fluctuates around ~7.5 ktps. Is is also very effective in controlling the recovery backlog, since the number of dirty pages is kept consistently in the 0.4–0.5-million range, while the redo length is also kept very low at 1–2 GB. The I/O metrics are very similar to the eager strategy, indicating that the same bottlenecks are present; however, because the page cleaner makes a deliberate selection of which pages to write and which not, the same I/O resources are utilized in a much more effective way.

Despite the improved performance, transaction throughput is still quite low, considering that it can reach almost 30 ktps if there is no write activity on the database (as indicated by the warm-up phase of the eviction-oriented strategy). Given this observation, it is possible that too many writes are being performed on the database, which unavoidably hurts read performance for page misses. To test this hypothesis, a second experiment with the oldest-LSN policy was performed, in which cleaner rounds are only executed at every 500 ms rather than continuously.

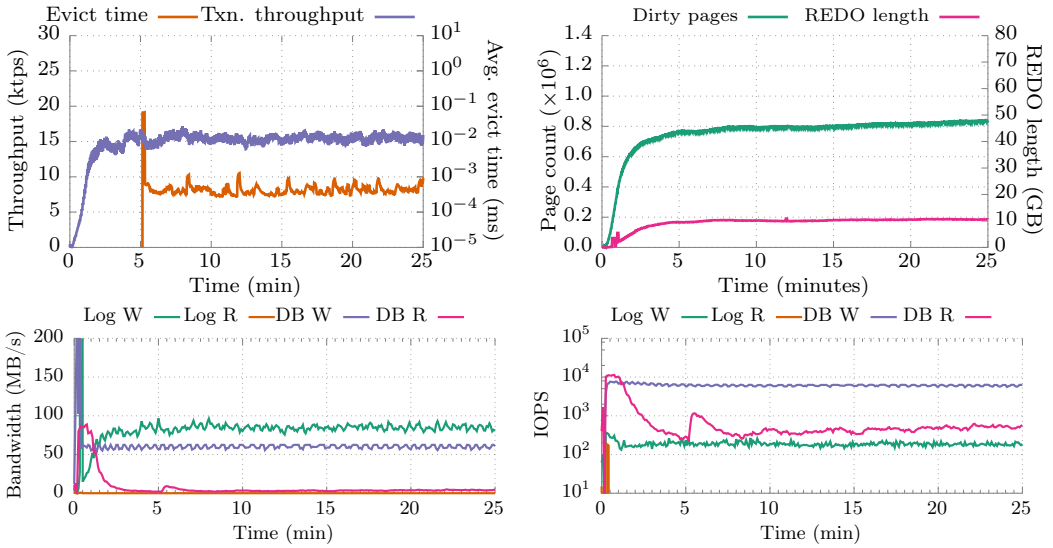

Fig. 49. Propagation efficiency and I/O metrics of page cleaning with oldest-LSN policy and 500 ms of sleep time between cleaner rounds

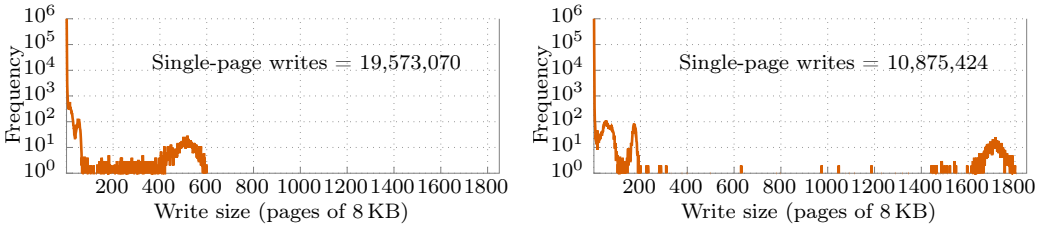

Fig. 50. Write size distribution of oldest-LSN policy without (left) and with (right) 500 ms sleep time

The results, shown in Figure 49 confirm this hypothesis: average transaction throughput doubles, reaching above 15 ktps. As the I/O metrics indicate, the reduced write activity results in an increase in read activity, i.e., the IOPS "saved" by the cleaner are directly reused to serve reads (note that the IOPS chart uses a logarithmic scale), resulting in improved performance. However, this comes at a cost on the recovery backlog: the dirty page ratio essentially doubles, while the redo length increases by a factor 5. This result clearly demonstrates the trade-off involved in managing recovery time and transaction throughput. On the upside, the employed policy is at least able to maintain a constant recovery backlog, thus avoiding the overflow situation described in the beginning of this chapter.

The improved write performance of the oldest-LSN policy is a direct consequence of the ability to exploit large writes. To verify this hypothesis, the size of each write performed by the page cleaner was collected and the distribution plotted in Figure 50. The left chart shows the results for the first experiment, with uninterrupted cleaner rounds, and the right chart for the second experiment, with 500 ms sleep time in-between rounds. Note that the y-axis uses a logarithmic scale. Since single-page writes still dominate, their absolute number is printed explicitly on the charts.

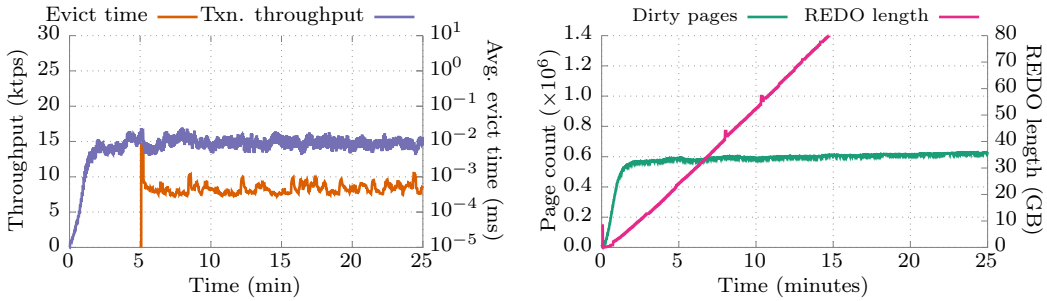

Fig. 51. Propagation efficiency of page cleaning with LRU policy and 500 ms of sleep time between cleaner rounds

Because of the lower frequency of cleaner rounds, the second experiment performs much larger writes, up to 1,800 pages, or 14 MB. Both charts display an interesting pattern, attributed to the characteristics of the TPC-C workload: a concentration of values towards the end range of the distribution. This is due to writes of pages belonging to order and history records, which are appended at a very high rate. As mentioned earlier, the ability to exploit this characteristic for efficient propagation is a good measure of page cleaner efficiency.

One potential concern with longer cleaner-round intervals is shown in the chart of the dirty page count, which seems to rise over time, albeit very slowly. This suggests that an overflow situation, as illustrated with the sink analogy earlier, might occur eventually. Without any sleep in-between rounds, on the other hand, this behavior is not observed. The conclusion is that in practice, the page cleaner service should adjust its sleeping time, or, in other words, its aggressiveness, according to system activity. One remaining open challenge is then to provide such adaptable behavior while delivering robust performance.

The last two experiments both considered the oldest-LSN policy only, but it would be worth to investigate the behavior of different policies. As mentioned earlier, there is a wide range of criteria to consider, as in page eviction strategies. In the scope of this thesis, multiple policies were implemented and evaluated, but presenting all of them here would be excessive without much added insight. Therefore, the following experiment considers one additional policy, which was observed to deliver similar performance to the oldest-LSN policy but deliver a different trade-off in terms of recovery backlog.

The next policy considered is equivalent to the traditional LRU, but considering updates only (i.e., least-recently updated). This policy selects the frames with the lowest page LSN value as candidates; it is also implemented in the PostgreSQL system [Pos], which combines it with eviction-oriented cleaning. Like the last experiment, a 500 ms interval between rounds is used. The results are shown in Figure 51; it omits I/O metrics since they are very similar to the oldest-LSN policy. The only advantage seems to be the lower dirty page count, which remains at ~0.6 million, in contrast with the ~0.8 million of the oldest-LSN policy. The redo length, on the other hand, grows very quickly, suggesting that many frequently-updated pages are never flushed, which is expected with the LRU policy.

One last experiment attempts to combine the oldest-LSN and LRU policies, by interleaving cleaner rounds using each policy. The results, in Figure 52, show that it combines the advantages of both policies: as expected, the dirty page count stays in-between 0.6 and 0.8 million, while the redo length is slightly larger than in the oldest-LSN policy, but still bounded.

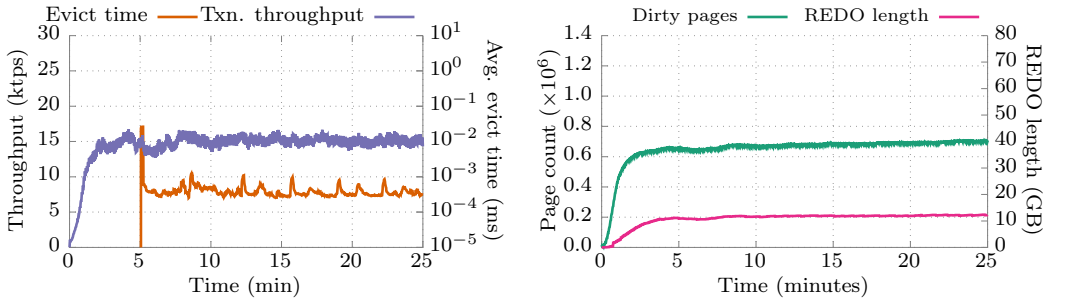

Fig. 52. Propagation efficiency of page cleaning with mixed oldest-LSN and LRU policy, with 500 ms of sleep time between cleaner rounds

4.3.4 Discussion

The evaluation of page cleaning techniques presented above permits multiple conclusions.

First, it shows how performing single-page writes can hurt performance indirectly, by increasing the time it takes for a worker thread to find an empty frame to fetch a database page or allocate a new one. Despite the use of policies that prioritize dirty pages according to various criteria and aggregate writes of adjacent pages, single-page writes still dominate. As the experiments also demonstrate, this is a concern even with SSD devices, because of the requirement of synchronous writes. Techniques that improve write bandwidth are discussed in the next section. The goal of the present section is to show that page-oriented propagation is not efficient enough to keep up with fast, memory-resident transactions.

Second, the algorithms presented earlier in this chapter and the different parameters considered in this evaluation show how tightly coupled propagation strategies are with the buffer pool. Keeping track of dirty pages and their recovery LSN is a major source of complexity and overhead. For example, the only reason why single-page writes must be synchronous is that their recovery and persisted LSNs must be updated in the buffer pool and a page-flush log record is generated after successful completion of the write. Transaction durability, i.e., the "D" of ACID, is guaranteed by the write-ahead log alone; thus, it seems like a major waste of resources to perform page writes in any way that is not fully asynchronous. This suggests that new recovery techniques, which relax the bookkeeping requirements for pages in the buffer pool, may contribute to more efficient (and easier to implement) page cleaning strategies.

Third, and perhaps most importantly in a more abstract level, the analysis presented here demonstrates that efficient propagation is hard to achieve, given the different optimization goals and the wide range of parameters that can be adjusted. The present evaluation only considered a small subset of policies and configurations, all under a single workload and with the same underlying setup. It is clear that an evaluation of all possible strategies would be excessive, and, most likely, no single configuration would outperform all the others across all scenarios. This makes the case for more robust propagation techniques, in which there are less knobs to turn and performance is actually better. The next section proposes one such technique.

4.4 Log-based propagation

The propagation strategies discussed so far involve flushing pages directly from the buffer pool into their permanent location on persistent storage, i.e., they are *page-based* propagation strategies. This has traditionally been an essential cornerstone of write-ahead logging systems [HR83]. This

section analyzes some of the benefits and drawbacks of this traditional design and its improvements proposed in related prior work. Furthermore, it proposes and evaluates alternative techniques that rely on the log as a propagation tool, thus expanding its utility from just recovery support.

4.4.1 Related work

The buffer pool component of a database system has two main concerns: caching and propagation. The previous sections already highlighted the separation between these concerns by distinguishing eviction—i.e., choosing which pages to cache in main memory—from cleaning—i.e., choosing which pages to propagate to persistent storage. While these concerns should be kept conceptually separate, in practice they are considerably intertwined and both rely on the buffer pool infrastructure. They are intertwined because eviction should avoid picking dirty pages as victims, and thus it should be able to request a cleaner round when it encounters too many dirty pages. Eviction can also directly invoke the write of a single page if it chooses to evict a dirty page. The page cleaning policy, on the other hand, can positively influence caching behavior by making sure that good eviction candidates are kept in a clean state. As the algorithms presented earlier in this chapter show, both services make use of buffer-pool control blocks to manage metadata and pick candidate pages. They also rely on latching and other buffer pool functionality for concurrency control.

For the reasons given above, addressing the concerns of page cleaning and eviction with a unified, centralized system component—the buffer pool—is a sound design decision. However, it has certain drawbacks that make it worthwhile to investigate alternative approaches, especially considering the high transaction volume enabled by memory-abundant servers with modern storage hardware. The next paragraphs discuss these drawbacks in detail.

One of the major performance problems of page-based propagation has already been mentioned in Section 4.2.1, which emphasized the reasons why it is important to implement efficient page cleaning strategies. Despite the range of choices for a page cleaning strategy, the basic approach of flushing pages directly from a buffer pool frame into their permanent location on disk has inherent limitations. In particular, it only provides a best-effort approach in terms of exploiting large writes; in practice, as shown earlier in Section 4.3, most page cleaner writes are of single pages, i.e., essentially random. Therefore, performing large writes is crucial to maximize cleaning throughput[14].

Related prior work has addressed this problem in multiple ways. Log-structured merge trees [OCGO96] perform propagation solely with large, sequential writes; another approach with similar behavior is a partitioned B-tree [Gra03a, Gra06a]. While these approaches provide the desired improved write bandwidth, they require a specialized index implementation to support the log-structured organization, i.e., they cannot be employed transparently for any access path. Write-optimized B-trees [Gra04] provide large sequential writes by migrating pages to new locations on every write. This approach has the advantage of not requiring algorithmic changes to index lookups and scans, but it requires index structures with a single incoming incoming pointer between a parent node and its child, in order to support page migration. This requirement is alleviated if a page identifier is mapped indirectly to its location on disk, as in the "database cache" approach [EB84] or with log-structured file systems [RO92]. However, unlike write-optimized B-trees, the indirect mapping adds overhead to index operations and the log-structured organization is imposed on all access paths, even those that would perform better with in-place updates.

[14]Modern SSD devices are able to deliver relatively high random-write bandwidth, but that is usually only achieved with multiple threads and fully asynchronous writes. For the reasons discussed in Section 4.2.3, a page cleaner service requires synchronization to mark pages as clean and it runs with a single thread (or one thread per physical partition of the database). Thus, the problem of single-page writes is not fully mitigated with modern storage devices.

While the approaches mentioned above address the problem of small writes during propagation, they still rely on flushing pages from main memory directly into their assigned location on persistent storage. As such, they still suffer from the other drawbacks mentioned below. Furthermore, despite the tendency of referring to such techniques as "log-structured", they do not reuse the write-ahead log (which is required anyway for transaction support) as a mechanism for (or at least as an aid to) update propagation.

Log-driven backups, proposed by Levy and Silberschatz [LS90], rely on the transaction log to replay updates on the materialized database[15]. However, in their approach, the log is replayed in its original LSN order, which means that single-page random writes are performed on the database, as in traditional page cleaning.

Another performance concern of page-based propagation is its constant interaction with the buffer pool, which may negatively impact the performance of critical operations such as a page fix. Page cleaning requires not only inspecting buffer pool blocks but also latching frames multiple times in its three stages—candidate collection, copying, and updating LSN values. This unavoidably causes interference with running user transactions; however minimal this interference may be, it should not be ignored, especially because it tends to become more pronounced as transaction execution is optimized for modern hardware. Furthermore, lower interference is also desirable because it allows propagation services to run more aggressively.

Besides page cleaning and eviction, another responsibility of the buffer pool, which can be classified under the category of propagation concerns, is checkpointing. As explained earlier in Section 2.3, traditional ARIES-based checkpoints require collecting a list of dirty pages. As with page cleaning and eviction, this requires inspecting buffer-pool control blocks, further increasing the interference mentioned above. A study of this interference in a prototype similar to the one used in this thesis was performed by Lersch et al. [LSH16].

Lastly, it is worth pointing out that page-based propagation also makes it hard to measure and reason about its actual effectiveness, considering all goals involved: reducing recovery time, recycling log space, enabling quick eviction, utilizing write bandwidth effectively, minimizing write amplification, etc. Given this complex space of policies and, to a large degree, conflicting optimization goals, simpler and robust techniques that eliminate design choices while delivering satisfactory performance are preferable.

4.4.2 Decoupling propagation concerns from the buffer pool

This thesis proposes a novel technique for update propagation that does not rely on the buffer pool, so that its only concern is caching. The basic idea, which is illustrated in Figure 53, is to rely on the log to perform all propagation tasks, including checkpoints and page cleaning. In this way, the buffer pool implementation is substantially simplified, since there is no need to keep track of the propagation state (i.e., persisted and recovery LSN) of each frame in the buffer pool. This separation or concerns also has a direct performance benefit, since it eliminates the interference caused by such book-keeping.

The key to achieve fully decoupled, log-based propagation is to rely on the techniques presented in Chapter 3, which provide a way to replay the log in page-ID order and to recover individual pages on demand.

[15]The term "backup" is employed by the authors in the context of early in-memory databases, and it is unrelated to media recovery. It simply refers to the materialized database, and thus it can be seen as a page cleaning technique.

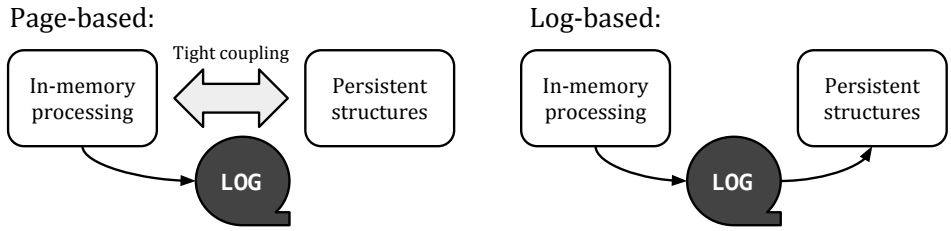

Fig. 53. Coupling of persistent and in-memory components

4.4.3 Log-based checkpoints

The first step in achieving decoupled propagation is to use log information to produce checkpoints, rather than scanning the buffer pool and the transaction and lock managers. This can be easily achieved by simply reusing the log analysis phase of restart recovery. The goal of log analysis is to determine the set of dirty pages and their associated LSNs (either expected page LSN or recovery LSN) as well as active transactions and, in instant restart, their acquired locks. It achieves that by scanning the log from the last checkpoint up to the log tail. Therefore, one way to think of log analysis is as a procedure that takes the information from an older checkpoint and updates it with the current (fuzzy) state by performing a log scan. This is exactly how log-based checkpointing works—it is a system service that continuously scans the log in the background, keeping track of dirty pages and active transactions and periodically saving that information as a new checkpoint.

Because it essentially reuses the log analysis algorithm, very little change is required to implement log-based checkpoints. Furthermore, no change is required on any recovery component, since the persisted checkpoint information is exactly the same as produced by traditional checkpoints.

One concern with log-based checkpoints is that they incur read traffic on the log device, but this can be mitigated in two ways. First, the checkpoint information can be maintained eagerly, by processing the log while it is still cached in main memory. Second, the actual log scan can be shared between log-based checkpoints and log archiving with run generation.

In terms of computational overhead, log-based checkpointing maintains lookup tables of pages, transactions, and (for instant restart) locks. These must be updated for every log record describing a transaction update, but this is performed outside the critical path of transactions, whereas previous methods require updating control blocks during a page fix. If log-based checkpoints are integrated with run generation, so that the log can be consumed in page-ID order, the dirty page table is only updated once for each page, rather than once for each log record.

4.4.4 Log-based page cleaning

Log-based page cleaning is a technique that relies on log replay to propagate updates to the materialized database. Rather than flushing a page in its current state directly from the buffer pool, it loads its older version from the database, replays logged updates on it, and saves the newer version back to the database. At first, this seems like a very wasteful procedure, given that the most recent version is already available in the buffer pool. However, the benefit of this technique comes from the fact that these operations are performed in bulk, in a way similar to the instant restore algorithm presented in Section 3.4.

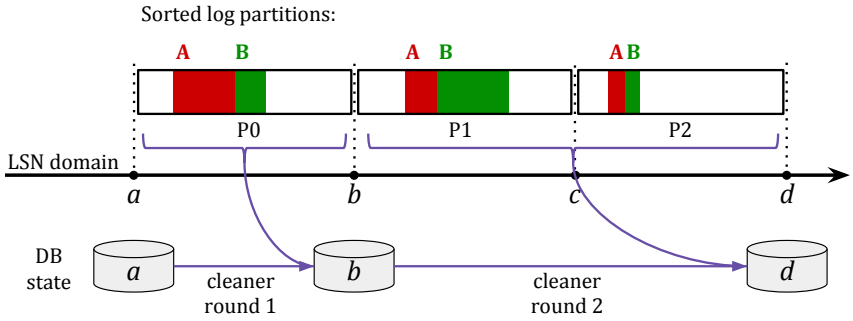

Fig. 54. Log-based cleaner rounds guided by sorted log partitions

Rather than replaying the normal recovery log (i.e., in LSN order)—as suggested in the approach by Levy and Silberschatz [LS90]—the log-based page cleaner relies on the partitioned log index introduced in Section 3.3. The cleaner algorithm is executed in rounds, as in the page-based counterpart, where each round consumes a sequence of sorted log partitions. Therefore, each round basically brings the materialized database from state a into state b, where a and b are the begin and end LSN of the first and last log partition consumed, respectively. This is illustrated in Figure 54, which shows the database being propagated from state a into b in a first cleaner round, and then from b into d in a second round. Note that multiple log partitions can be replayed at once by merging.

The log-based cleaner performs reads and writes on the database at a *segment* granularity, i.e., a set of contiguous pages. Therefore, the log replay pattern is identical to that of instant restore. Unlike instant restore, however, there is no need to control the propagation state using a segment recovery bitmap, because running transactions can freely access the database in the midst of a cleaner round. Nevertheless, a bitmap can be useful to avoid replaying the same segments multiple times in case of a system failure. Another difference to instant restore is that the index information on the log is not strictly needed, since individual log partitions are scanned sequentially, as in single-pass restore.

Each round of the log-based cleaner starts on the LSN on which the last round stopped. This LSN must be the beginning of a log partition, since it is the minimal unit of propagation. The page ID of the first log record scanned determines the first segment to be fetched from the database. Segment borders can be either specified dynamically, by considering the page ID of the first log record to be the beginning of the segment, or they can be statically mapped to fixed partitions of the database. Log records are replayed on the fetched segment until the next incoming log record belongs to a different segment, in which case the new segment must be fetched into the cleaner buffer.

The cleaner buffer should be larger than a segment, so that multiple segments can be written with a single write request. This essentially maintains a segment as a unit of read and log replay, but not as a unit of write. This can be beneficial because while large segments improve write performance, they increase the ratio of pages that are fetched and written back without any update being replayed on them. Note that because only segments for which there are log records in the log partitions being replayed are fetched, a smaller segment size decreases this ratio. Therefore, it is crucial to find a segment size that delivers a good trade-off between write performance and propagation granularity. This trade-off is also present in database warm-up strategies that fetch multiple neighboring pages to fill up the buffer pool [PDTP16].

If the cleaner buffer becomes full when switching to a new segment, the buffered segments are written back into the database. Each group of neighboring segments in the cleaner buffer is written with a single I/O call. After that, the control block of each flushed page must be updated to reflect the new persisted state, but only if that page is actually cached in the buffer pool. Because a page is not copied from the buffer pool to be cleaned, the page-based algorithm given in Section 4.2 does not apply, but the same infrastructure can be reused. With log-based cleaning, the persisted LSN can be obtained from the pages cleaner buffer, but since the cleaner has no prior interaction with the buffer pool, it does not know what the new value for the recovery LSN field should be. However, because the recovery LSN is only used for log space recycling and to specify the redo low-water mark for an ARIES-based redo log scan during restart, it can simply be set directly as the LSN on which that cleaner round will finish. This mechanism essentially establishes the LSN covered by a cleaner round as the new redo low-water mark, without the need to collect recovery LSNs from buffer pool frames when taking checkpoints.

One drawback of the log-based cleaner algorithm discussed so far is that it is still not completely decoupled from the buffer pool, because after flushing a segment, the corresponding control blocks must still be updated to reflect their new propagation state—i.e., to possibly mark them as clean. Therefore, it does fully not solve the problems mentioned in the beginning of this section—the buffer pool implementation still deals with propagation concerns, page cleaning still interferes with fix operations, checkpoints are still required to keep track of dirty pages, and, finally, transaction progress can still be slowed down with too many dirty pages in the buffer pool. To address these problems, an extended version of the log-based cleaner is presented below.

4.4.5 Log-based cleaner with write elision

Write elision is a technique that builds upon single-page recovery infrastructure to reduce the amount of pages in an incremental backup [GGS16]. In the context of log-based page cleaning, especially considering the goal of completely decoupling it from the buffer pool, the technique can be used to eliminate the need to maintain propagation state in the buffer pool—i.e., to eliminate the persisted and recovery LSN fields from control blocks.

The main idea of write elision is to use the page recovery index, introduced in Section 3.3, to enable any page to be evicted, regardless of its clean or dirty state, without writing it back to the database first. If that page is fetched again during a fix operation and its page LSN does not match the value stored in the page recovery index, then the missing updates are replayed using the per-page log chain. Note that the index is not updated on every page modification, but only when a page is evicted from the buffer pool. In the context of log-based page cleaning, per-page log replay is bounded by the LSN in which the last completed cleaner round finished—i.e., the redo low-water mark.

For purposes of recovery from single-page failures, the page recovery index is maintained permanently—possibly embedded in parent-to-child B-tree pointers [GKS12b]. However, if single page failures are not a concern, then the page recovery index can be cleaned up periodically as the cleaner completes each round, by deleting each entry whose expected page LSN is lower than the LSN covered by that round.

The use of write elision unleashes the full potential of log-based page cleaning as a decoupled propagation strategy, freeing the buffer pool from all propagation concerns. Also note that the use of a page recovery index permits checkpoints that only track active transactions, without any dirty page information; this is because the list of dirty pages, their expected LSNs, and the redo low-water mark are all embedded in this index.

One potential concern with log-based propagation with write elision is how it can control the recovery backlog and avoid the overflow situation mentioned in the beginning of this chapter. With traditional page cleaning, and also in the log-based cleaner without write elision, transaction progress is always slowed down if the redo length grows too large or there are too many dirty pages in the buffer pool. Thus, the tight coupling with the buffer pool is actually an advantage here, since it provides a natural throttling mechanism for when propagation cannot keep up.

In the log-based page cleaner with write elision, multiple mechanisms can be implemented to detect—and possibly avoid—an overflow situation. In terms of recovery backlog, the only concern in this case is the redo length, since there is no tracking of dirty pages in the buffer pool anymore; the number of pages that require redo in case of a system failure is simply the number of pages for which log records were produced since the LSN covered by the last cleaner round. Therefore, proactive throttling measures can be employed whenever the redo length grows too large.

One additional technique that can help in controlling the recovery backlog with log-based page cleaning is to employ page-image log records, which were introduced in Section 3.1.3. These essentially provide a flexible way to trade-off database bandwidth for log bandwidth: if a log record containing the full contents of a page is produced, then the propagation of that page can be delayed to a future cleaner round. This can be useful, for instance, if a segment being processed by the log-based cleaner only contains log records of a few pages; in that case, page-image log records can be generated and their propagation to the database is delayed to a future round, when hopefully more pages will be replayed in that segment. Note that this is very similar to the original motivation behind write elision, namely the reduction of incremental backups [GGS16].

Finally, one major advantage of log-based cleaning with write elision is that it can easily be combined with traditional page cleaning, using any of the techniques presented earlier in this chapter. In fact, those techniques become much more attractive, since the need to maintain propagation state in the buffer pool is eliminated, which opens the possibility for fully asynchronous single-page writes. Page-oriented cleaning can also be mixed directly in the log-based algorithm. In the situation mentioned above, for instance, rather than generating a page-image log record, if the given page is cached in the buffer pool, it could be simply flushed to the database directly. In conclusion, many such tricks and techniques are possible under this new paradigm of decoupled propagation; however, covering and evaluating them all is out of the scope of this thesis.

4.4.6 Evaluation of log-based cleaning

In order to evaluate the performance of the log-based page cleaner, the same experiment of Section 4.3 was executed with the two basic variants of log-based cleaning discussed above: the standard one, in which propagation state is maintained in control blocks of the buffer pool, and the one with write elision, which is completely decoupled from the buffer pool. The segment size chosen here is 64 pages of 8 KB, or 512 KB in total. This size delivered the best results in similar experiments, so other sizes are not considered here.

Results for the standard cleaner are shown in Figure 55. The first noticeable effect is the wide and periodic fluctuation in transaction throughput from the ten-minute mark onwards, which is directly correlated with the average eviction time. This behavior is expected because of the periodic behavior of the cleaner: it replays segments until its workspace fills up, after which writes are issued for groups of contiguous segments. During these periods of intense write activity, database reads slow down considerably, which explains the dips in throughput. Another cause for this effect is the accumulation of dirty frames in the buffer pool, shown in the top-right chart. After the cleaner

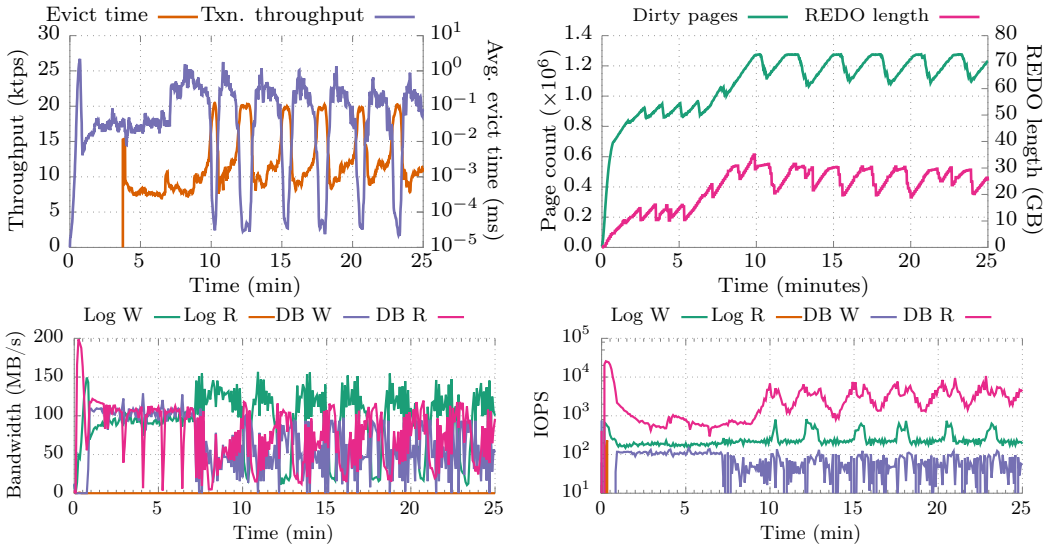

Fig. 55. Propagation efficiency and I/O metrics of log-based page cleaning

flushes its buffered segments, it updates the persisted LSN values in the buffer pool, thus reducing the number of dirty pages and reducing the average eviction time.

The different behavior of this experiment before the ten-minute mark is due to an initial accumulation phase, in which cleaner rounds process fewer sorted log partitions. For example, the first round is activated after the first partition is produced, but by the time the second round starts, three or four partitions may be available, and so on. In the experiment shown here, eighty partitions of 120 MB are replayed per round in the steady state, i.e., after the ten-minute mark.

The standard log-based cleaner is also not better than page-based cleaning in terms of recovery backlog. As the top-right chart of Figure 55 shows, both the dirty-page ratio and the redo length are higher than in the page-based cleaner with oldest-LSN policy. However, the stability of these values is an advantage here—despite the "zigzag" pattern, which is expected with the periodic, bulk updates of buffer-pool control blocks, these values do not seem to grow over time.

Overall, the results observed for the standard log-based cleaner are not convincing. Page-based cleaning with appropriate policies delivers much more stable behavior and lower recovery backlog. The next experiments evaluate log-based cleaning with write elision, which is expected to overcome some of the problems with the standard log-based cleaner.

The results for log-based cleaning with write elision are shown in Figure 56. Here, transaction throughput also fluctuates, but less drastically; in this case, however, the fluctuation does not correlate with average eviction time. This is expected, because eviction is never blocked by the page cleaner, i.e., pages are evicted regardless of whether they are clean or dirty—in fact, the buffer pool does not even know if a given page is dirty or not. In this experiment, the dips in throughput are caused solely by the intense write activity on the database when the cleaner flushes its buffer; this is clearly shown in the I/O bandwidth chart on the bottom-left quadrant.

Perhaps the biggest advantage of the write elision technique observed in this experiment is how well it controls the recovery backlog. The redo length is kept at a minimal level, which is expected since propagation is now guided by the log; thus, as long as the cleaner can keep up with transaction

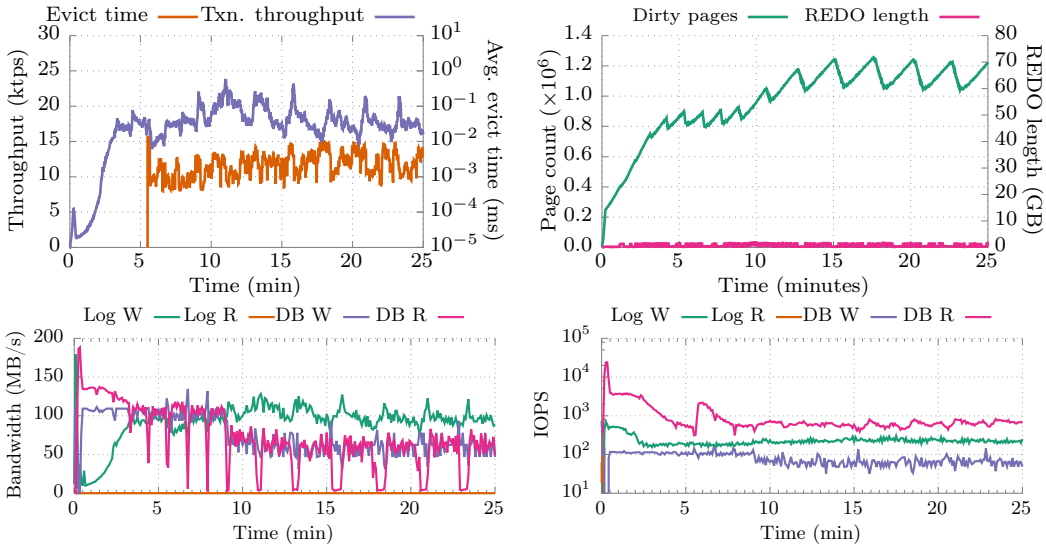

Fig. 56. Propagation efficiency and I/O metrics of log-based page cleaning with write elision

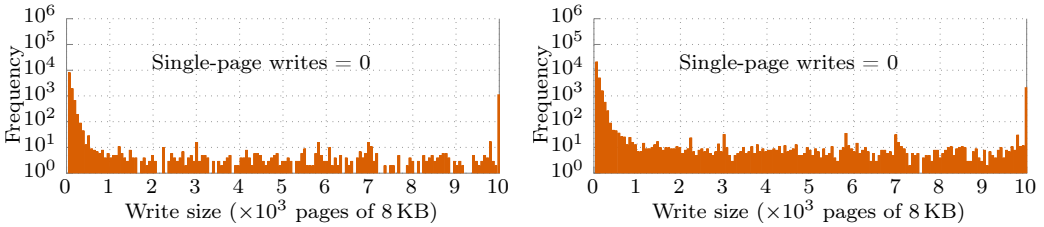

Fig. 57. Write size distribution of log-based cleaner without (left) and with (right) write elision

activity, a backlog of log partitions to be cleaned should not accumulate. The number of dirty pages in the buffer pool, on the other hand, is not better than with the standard cleaner and also worse than the page-based strategies. However, it is important to note that, with write elision, there is no dynamic dirty page tracking as with the other strategies. What the chart plots here as the number of dirty pages is simply the number of entries in the page recovery index, which is what is saved as a list of dirty pages during checkpoints. Because the page recovery index is updated whenever a page is evicted, even if that page is in fact clean, the number of entries is indeed expected to be much higher. Therefore, the number of dirty pages plotted here is not to be considered a direct measure of recovery effort in case of a system failure.

Lastly, the charts in Figure 57 show the distribution of write sizes for the two variants of the log-based cleaner: without (left) and with write elision (right). As expected, there are no single-page writes and all writes are multiples of the segment size, which is 64 pages. Note that the x-axis is displayed in units of one-thousand pages. Write sizes here are much larger than with page-based cleaning, up to the maximum 80 MB, which is the workspace size. In fact, as the distribution shows, such maximum writes are almost as frequent as those of the smallest sizes. Both variants exhibit very similar distribution, but thanks to its higher throughput, the write-elision variant performs more writes overall.

4.5 Summary of propagation strategies

This chapter explored the challenge of providing efficient propagation for intensive transactional workloads. Techniques for asynchronous page cleaning in the buffer pool were presented and different strategies were evaluated. The evaluation considered a buffer pool size that barely fits the benchmark's working set. This should be the most challenging situation for efficient propagation, because memory is not so small that transaction throughput would be I/O-bound, but also not so large that propagation can be delayed arbitrarily. Even if very large memory is considered, efficient propagation is still very important for availability, as it controls the amount of recovery work that must be performed in case of a system failure—referred to here as the recovery backlog.

Existing techniques either compromise on transaction performance or are not able to keep the recovery backlog under control. As a novel, alternative solution, this chapter presented log-based propagation strategies, which build upon the partitioned log index data structure introduced for instant recovery. Log-based strategies attempt to decouple propagation concerns from the buffer pool, thus providing more efficient I/O behavior with less interference on running transactions. The empirical results confirm the feasibility of the approach in practice, while demonstrating an incremental improvement over page-based techniques. As the analysis reveals, despite the visible improvements provided by the decoupled architecture, controlling I/O traffic to the database remains the key challenge, as bursts of intensive write activity unavoidably disturb transaction throughput if the amount of main memory is restricted. Therefore, future improvements over the techniques presented here should seek more effective ways to maintain *propagation* efficiency with less aggressive *write* activity.

The problem of efficient propagation becomes even more challenging if other optimization goals, which are quite relevant in practice, are incorporated, such as write amplification. Write amplification is a measure of how often a certain data item is rewritten on persistent storage in comparison to how often it is actually updated by the application. The techniques presented here all write to the database very aggressively, which is undesirable in practice, especially with devices that have low endurance, such as flash memory. Therefore, the work presented in this chapter leaves much room for future improvements.

The main take-away message of the study presented here should be that efficient propagation is a difficult and important challenge for database system architects and programmers. Building upon some of the ideas investigated here, the next chapter presents a novel database system architecture that should drastically simplify the implementation of efficient propagation strategies.

5 THE FINELINE LOG-STRUCTURED DATABASE

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 chapter proposes a novel approach for transactional storage and recovery called FineLine. It simplifies the implementation of transactional database systems by maintaining all persistent data in a single, log-structured data structure. This approach not only promises more efficient recovery with less overhead, but also decouples the management of persistent data from in-memory access paths. In fact, FineLine introduces a new, generalized perspective to data storage and recovery, in which the distinctions between main-memory and disk-based database systems are essentially blurred.

This chapter aims to be independent of the rest of this thesis, so that the concepts herein can be understood without first reading the previous chapters. Therefore, some of the concepts introduced earlier will be repeated here. Where appropriate, relevant details on previous chapters will be referred to. Unlike the previous chapters, the FineLine approach was not evaluated empirically in the context of this work. Nevertheless, this chapter discusses implementation details and elaborates on the performance expectations of these techniques.

5.1 Motivation

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.

The primary reason behind such a *dual-storage* design, illustrated in Figure 58a, 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*.

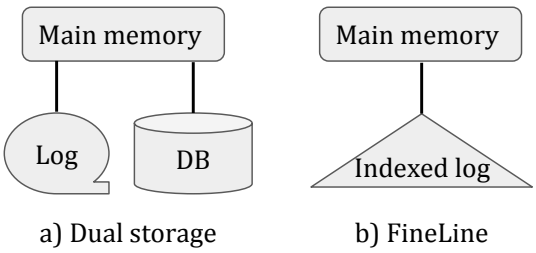

Fig. 58. Single-storage principle of FineLine

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, as explored in detail in Chapter 4 of this thesis, the propagation of updates into the read-optimized database may not only slow

down transactions but also easily lag behind the log, leading to an unbounded growth of the log and thus of recovery times.

These problems are known for a long time, as noted recently by Stonebraker, referring to the now 30-year old POSTGRES design [Sto87]:

> *"…a DBMS is really two DBMSs, one managing the database as we know it and a second one managing the log."*
>
> Michael Stonebraker [Sto16]

A *single-storage* approach eliminates these problems, but existing designs sacrifice either: (i) efficiency, by requiring random writes during commit processing, leading to lower transaction throughput; (ii) generality, by requiring specialized hardware such as byte-addressable non-volatile memory; or (iii) reliability, by decreasing redundancy and/or incurring very long recovery times.

This chapter 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 Figure 58b.

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 varying degrees of working-set to memory ratios. Therefore, the main goal of FineLine is to provide a generalized software architecture to implement transactional storage on top of traditional as well as modern hardware platforms in a unified way.

In the remainder of this chapter, Section 5.2 discusses related work. Section 5.3 introduces the basic system architecture and its components. Sections 5.4 and 5.5 then expose the details of the logging and recovery algorithms, respectively. Finally, Section 5.6 discusses implementation aspects and Section 5.7 concludes this chapter.

5.2 Related work

This section discusses related work, focusing on the limitations that the FineLine design aims to overcome. The FineLine 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 design.

5.2.1 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.

Approaches that completely eliminate the log must employ a *no-steal*/*force* mechanism [HR83], 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 [BHG87, HR79, Lor77], but they never made it into production use because of their inefficiency in traditional storage architectures. The POSTGRES system [Sto87] 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 [DKO+84, MWMS14, TZK+13], which eliminates persistent undo logging, but a log is still required for redo recovery.

The LogBase approach [VWA+12] 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. In 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) [OCGO96].

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 log 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 [MKM12] used in Silo [TZK+13] or the Adaptive Radix Tree [LKN13] used in Hyper [KN11].

The third limitation is that recovery from a system failure 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 [VWA+12], it seems like traditional recovery techniques like ARIES [MHL+92] or System R [GMB+81] are required.

5.2.2 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 [MHL+92] 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 [HR83], 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 [MWMS14]. 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 [HR83]—and maintaining it requires expensive checkpointing procedures that rely on shadow paging [GMB+81] or copy-on-write snapshots [MWMS14, KN11]. The CALC approach [RDAT16b] 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 [RDAT16b]. 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 [MWMS14].

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 [MWMS14]. 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 [TZK+13], scalable log managers [JPS+12], and distributed logging [WJ14, TZK+13]. 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.

5.2.3 Instant recovery

Instant recovery [GGS16], which was discussed and evaluated in Chapter 3 of this thesis, 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 main instant recovery algorithms—instant restart after a system failure and instant restore after a media failure—exploit the independence of recovery among objects that is inherent to physiological logging [MHL+92]. 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 [YAC+16] 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" [PDTP16].

5.2.4 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. While these efforts seem promising, assuming that the new technology will replace all other forms of storage is likely unrealistic—at least in the short to medium term. A relevant example is the advent of flash memory: rather than completely replacing magnetic hard disks, they are 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 fact that many production systems now support SSD extensions to their traditional disk-based architecture [DZP+11, CMB+10] is strong evidence to this expectation.

The "five-minute rule", introduced three decades ago [GP87] and refined multiple times since then [GG97, Gra09b], points to the importance of considering economic factors when planning memory and storage infrastructures. The emergence of new memory technologies is likely to further promote the importance of this rule, leading to a wider spectrum of choices and optimization goals. Therefore, systems that embrace the heterogeneity of storage hardware with a unified software approach are desirable.

5.2.5 Log-structured storage

Much like the recovery log in a write-ahead logging approach with a no-force policy [HR83], an LSM-tree [OCGO96] 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 [SLHG16]; during restart, at least in a traditional ARIES-based approach [MHL+92], 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.

Previous work on instant recovery from media failures [GGS16] 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 [SGH17] 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 [CDG+08] to provide ACID semantics. Given that the log itself can be reorganized and indexed, as done in instant recovery [GGS16], 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 log-structured merge trees to store and organize log records seems to be an effective way to eliminate database storage and provide a single-storage transactional system.

5.2.6 Summary of related work

In the wide design space between traditional disk-based and modern in-memory database systems, four major design goals are identified 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. Furthermore, log-structured techniques are considered 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.

5.3 Architecture

Figure 59 shows the four main components of FineLine. This section provides an overview of these components and their interaction.

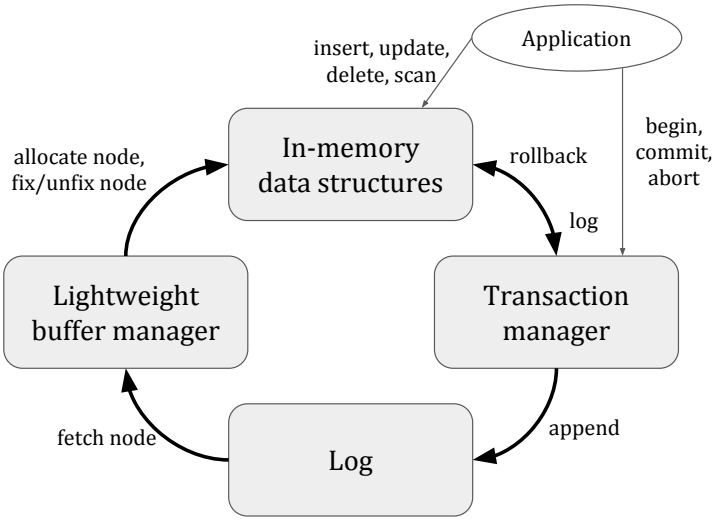

Fig. 59. Overview of the FineLine architecture

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.

5.3.1 In-memory data structures

FineLine is designed as a generic key-value store for a wide variety of transactional applications. It provides a simple application interface that allows operations on single keys, such as *get*, *insert*, *delete*, and *update*, as well as range scans. Therefore, 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 persistent 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 Figure 60, 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; left) and in FineLine (right). A typical WAL-based database system propagates changes to persistent storage by flushing a node of a data structure into a corresponding page on disk. Because this propagation happens asynchronously (i.e., following a no-force policy [HR83]), 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 or any other part of an in-memory data structure. Instead, it relies on the log, which is indexed on node identifier

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 5.3.2 and 5.4 below discuss the details of the FineLine log, referring back to Figure 60.

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, as in the case of a traditional B-tree, 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.

Depending on the data structure implementation, a node can be anything from a single record to a whole database. This generalized model introduces a continuum of system designs, each with its own tradeoffs and implications for performance, recovery, and availability. On one end of the spectrum, if the node granularity is chosen to be the whole database, the recovery algorithm behaves essentially like the command logging approach [MWMS14]; whereas a behavior similar (but in many ways superior) to ARIES [MHL+92] is achieved if a node corresponds to a page.

5.3.2 Log

The log component is the centerpiece of the FineLine design, and the main contribution of this work. As Figure 59 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 [OCGO96, SR12]. Internal aspects of the log are discussed in Section 5.4; this section focuses on the interface to the other components through the append and fetch operations, as illustrated in the right side of Figure 60.

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 5.4.

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.

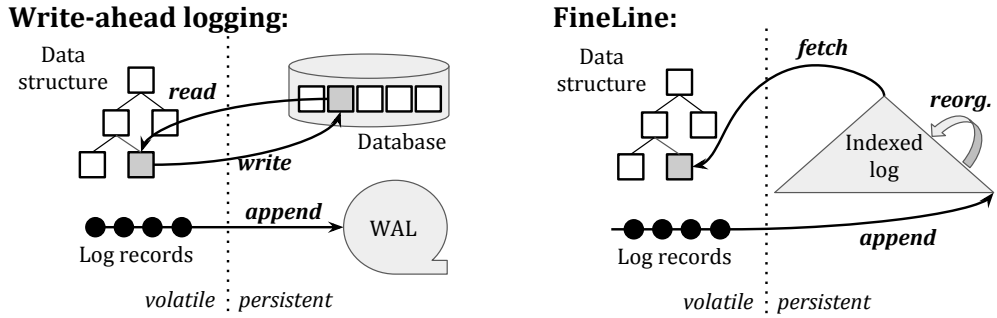

Fig. 60. Propagation in WAL (left) vs. FineLine (right)

5.3.3 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 [GR93]—preferably optimized for main memory [GVK+14]—as well as record-based caching schemes [DPT+13, ELL14] 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.

5.3.4 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 5.4 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 [KN11, MWMS14], a transaction-private undo log is used for rollback and it may be discarded after commit[16]. 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

[16]Systems with optimistic concurrency control (e.g., Silo [TZK+13]) must keep track of the *write set* of each transaction, which is in essence a transaction-private log—the only difference is the recording of after-images instead of before-images.

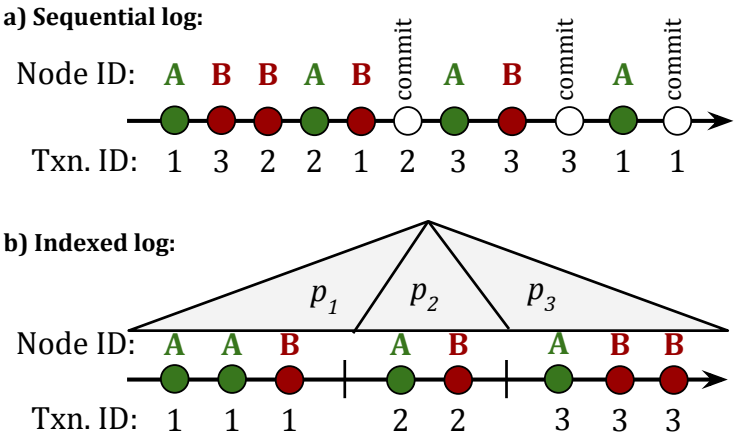

Fig. 61. A sequential log (a) and its equivalent indexing (b)

in Figure 59). In an optimistic concurrency control scheme, the transaction manager also performs validation in its commit protocol. The present design does not assume any specific concurrency control scheme; thus, the transaction manager is treated as an abstract component.

5.4 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.

5.4.1 Indexed log organization

The indexed log is actually a partitioned index, in which probes require inspecting multiple partitions before returning a result [Gra03a]. 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 5.1, 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.

Figure 61 contrasts the partitioned-index log organization used in FineLine and 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

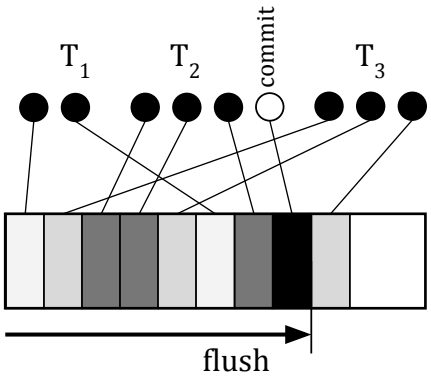

Fig. 62. ARIES commit protocol

absence of undo recovery, as discussed in Section 5.5. For a comprehensive discussion on the issue of log record ordering, refer to the work of Wang and Johnson [WJ14] and their GSN approach.

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 [OCGO96, RO92, SR12], 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 5.4.3 provides further details of the fetch operation and the reorganization of log partitions. Before that, Section 5.4.2 discusses how new partitions are created with transaction commits.

5.4.2 Commit

Commit processing in ARIES, shown in Figure 62, 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 and resources can be freed—the transaction may not rollback anymore and the 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[17]. 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[18]. For ease of exposition, it is assumed 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 [JPS+12] and also used in the Silo in-memory database [TZK+13], essentially implements the group commit approach [DKO+84], 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 [TZK+13]. 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 Figure 63 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, a single, system-wide log buffer is assumed, but the technique can be extended to multiple commit service threads to provide distributed logging. Such extension is out of the scope of this thesis, but existing techniques can be adapted [TZK+13, WJ14].

Note that because the log is an index, appending a new partition also requires inserting a pair of separator key and child pointer in the parent page; this may, in turn, require a split. In order to eliminate this overhead from the commit critical path, empty leaf pages are pre-allocated, formatted, and inserted into the index asynchronously using system transactions. In essence, the commit service must only flush a single log page into a previously allocated slot on persistent storage—any index maintenance or reorganization is performed by asynchronous system transactions.

5.4.3 Node fetch and merging

In the architecture diagram of Figure 59, 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 *page-image* log record is found, which can be either an initial node

[17] The choice of log page size depends on a good balance between maximizing write bandwidth, minimizing fragmentation, and optimizing group commit schedules [HSL+87].

[18] This is the approach proposed in Section 3.3 of this thesis.

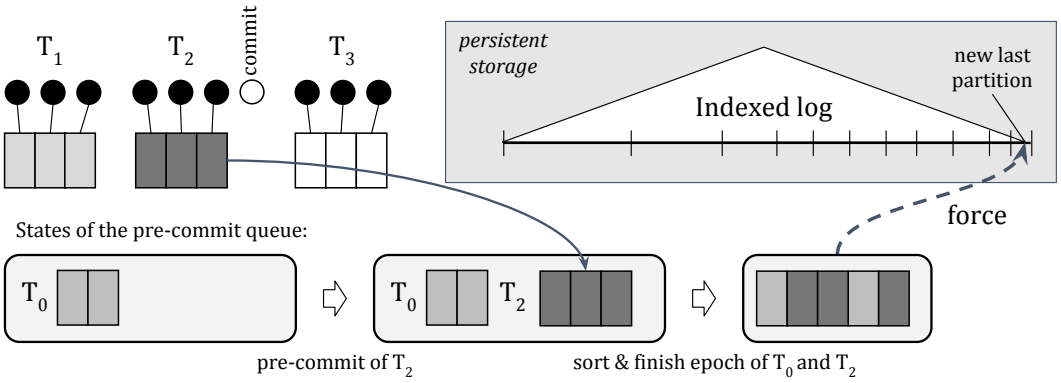

Fig. 63. FineLine commit protocol

construction and allocation or a "backup" image of a node, containing all data required to fully reconstruct it. Then, starting from the page-image 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 merged and compacted periodically with an asynchronous daemon, like in LSM-trees [OCGO96, SR12]. 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.

Figure 64 illustrates the merge process, where the two states (c and d) of the index are derived from the states (a and b) of Figure 61. 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. 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.

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.

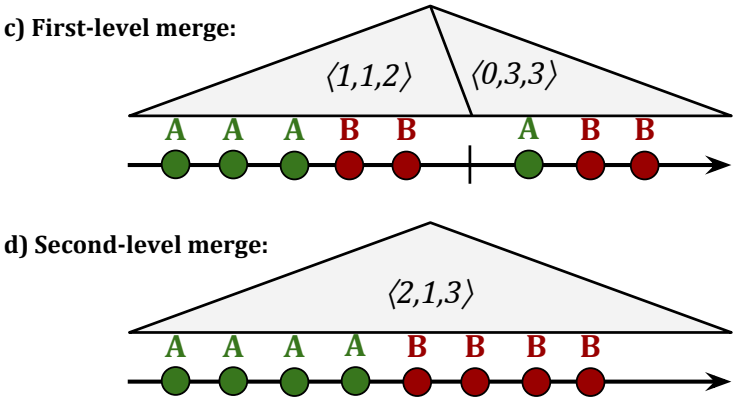

Fig. 64. Partitioned log after one (c) and two (d) merges

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 or zone indexes can be used to avoid probing partitions that are known to not contain a given node identifier [Gra09a]. 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. The problem of compaction can be characterized as follows: given a stream of log records $l_1 \ldots l_n$ pertaining to the same node, produce a stream of log records $k_1 \ldots k_m$ 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 page-image log records. If any of l_i is a page-image log record, then a single page-image 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. Page-image 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 dual-storage 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 page-image log record. Such higher-level partitions, containing one page-image log record for most nodes, can be viewed as a generalized form of database storage as employed in a dual-storage architecture.

5.5 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 [HR83]. 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—log 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.

5.5.1 Indexed log consistency

As discussed in Section 5.3, 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 chapter. 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 [APD15, Rod08] 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 [APD15].

5.5.2 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) [MHL+92]. 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 [GGS16]. 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 [PDTP16]. 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 overhead 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.

5.5.3 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 media. Given that the "head" of the sequential log is expected to be recycled quickly as transactions are finished and the updates are propagated to their final destination in the database, the log device can be fairly small, and thus 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 thesis, the techniques involved are briefly discussed. In ARIES, media recovery essentially requires archive copies of both the log and the database [MHL+92]. 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 [SGH17] 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.

It is important to 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 [SGH17], 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.

5.6 Implementation

This section discusses two basic approaches to the implementation of FineLine: first, it can be adapted from an existing system built around the ARIES framework; second, it can be implemented from scratch. The following sub-sections discuss the details, advantages, implementation effort, and caveats of each approach.

5.6.1 Incremental implementation from ARIES

A transaction-processing system supporting the original ARIES specification [MHL+92] can be extended to support FineLine with a sequence of incremental steps. Many of these steps would be based on individual techniques of instant recovery [GGS16] and have also implemented and evaluated in Chapters 3 and 4 of this thesis.

The first incremental step would be the implementation of a sorting and indexing infrastructure that builds the indexed log from log records in the recovery log, exactly as described in Section 3.3 of this thesis. This data structure and its maintenance operations (e.g., creation, merging, garbage collection, etc.) provide everything that is needed for the FineLine log. However, unlike proposed in Section 5.4, the recovery log would be the last unsorted partition, and per-page chains of log records are required to retrieve the history of each page (or, in the general nomenclature of FineLine, each node).

The second incremental step would be the implementation of the *write elision* technique [GGS16], as described in the log-based propagation strategy presented in Chapter 4. With write elision, every time a page is evicted from the buffer pool, its page LSN is added to the page recovery index. When that page is fetched again, its potentially out-of-date image is loaded from the database and the missing updates are replayed from both the recovery log or the sorted log partitions, exactly as described in Section 3.3.

The third incremental step would be simply the elimination of database storage. In the existing system, this could be achieved by simply never writing any page to the database, i.e., by disabling page cleaning completely. To minimize changes in the buffer pool component without unnecessary I/O calls on the empty database, a "dummy" wrapper class can be implemented to replace the system volume or device manager, by simply delivering an empty page on every read call. This empty page would contain a null LSN value, which would result in the replay of the page's history from its latest page-image log record to its most recent update. This essentially implements the *fetch* operation of FineLine.

By implementing the three major steps described above, the new system is already made single-storage and thus supports the basic FineLine principle. However, it retains the traditional commit protocol of ARIES and still requires both redo and undo recovery in case of system failures. Furthermore, the new system would not support distributed logging, which is natively supported in the design presented in this chapter. Previous work explored some ideas to support a no-steal policy in ARIES [SH14], but these are usually very cumbersome to implement because of logical undo with compensation and record-level locking. Furthermore, even if made to work, such solution would still require much of the machinery associated with undo recovery—checkpoints with active transactions and possibly acquired locks, compensation log records, transaction-related information in log records, etc.

Lastly, it is worth mentioning that an incremental improvement over ARIES that aims to provide distributed logging in multi-core machines was proposed by Wang and Johnson [WJ14]. Despite being targeted at non-volatile memory, their commit protocol and recovery techniques could be adapted in a hardware-independent system as well. The problem with their approach is that despite the performance improvements, the system architecture is made substantially more complex than traditional ARIES. Unlike FineLine, it does not eliminate checkpoints and undo recovery, does not support instant recovery, and does not decouple in-memory data structures from persistence concerns.

For the reasons given above, only the three incremental steps described earlier are worth incorporating into an ARIES system, turning it into a single-storage, log-structured system. The remaining features of FineLine are thus better supported if implemented from scratch, as discussed next.

5.6.2 Implementation from scratch

The main challenge to be addressed in a FineLine implementation from scratch would be to provide a generic buffer manager and logging interface to support arbitrary access paths. The lightweight buffer manager is only concerned with caching, and thus it would be much simpler to implement than a traditional database buffer pool. It should support pointer swizzling [GVK+14] natively, but without assuming any internal structure on individual nodes. An ideal programming technique to achieve that would be generic smart pointers, similar, for instance, to `shared_ptr` in the C++11 specification [Fou]. This would be a generic wrapper class that controls swizzling—i.e., the conversion between virtual memory pointers and node identifiers—in an access-path-independent way. Note that the FineLine design also makes it easier to implement pointer swizzling, since nodes are never directly flushed to persistent storage. Lastly, to support unswizzling during eviction, the lightweight buffer manager needs a way to iterate over pointers inside a non-leaf node in a generic way.

The requirements listed above essentially establish an API, or, in other words, a contract, between access path implementations and the lightweight buffer manager. The same holds for the log manager, which must provide logging, commit, and fetch (i.e., log replay) functionality in a generic manner. Luckily, such interface is also simple to implement in with the decoupled design of FineLine. Log records are managed by the log component as arbitrary, uninterpreted byte payloads associated with a node identifier and a version number. In other words, the log essentially provides a key-value store for log records. When fetching a node, individual log records are replayed using a callback function, which is also part of the API offered to access path implementations. Specific optimizations such as page-image log records require an appropriate extension of this interface.

The transaction manager, which was presented in Section 5.3 as one of the major architectural components of FineLine, requires much less interfacing with in-memory access paths, and thus it can easily be adapted from an existing system. This is mostly because transaction managers deal mostly

with logical identifiers that are unrelated to physical data representation, such as transaction IDs and lock IDs (note that, as mentioned earlier, this chapter considers lock management a sub-component of the transaction manager). The no-steal protocol of FineLine also eliminates many concerns of a traditional transaction manager, such as checkpointing. Despite the possibility of reuse, an important feature that the transaction manager must support is the management of transaction-private log buffers and their rollback. Depending on the implementation, this concern could also be delegated to the log component.

5.7 Summary of FineLine

This chapter 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 much of the overhead 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 practically instantaneous. 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.

6 CONCLUSION

This thesis provided a modern analysis of transaction-oriented database recovery techniques, focusing on the new challenges brought up by recent hardware developments. New techniques for efficient recovery and propagation were proposed and analyzed, aiming for gradual and incremental improvement over established, time-tested techniques, such as the write-ahead logging paradigm and the ARIES family of recovery algorithms. Over the course of the three main chapters, which are summarized below, incremental design improvements were proposed, gradually leading to the FineLine design, which is a radically new approach to transactional storage and recovery.

After a comprehensive review of background theory and related work, Chapter 3 of this thesis presented an implementation and evaluation of instant recovery techniques based on write-ahead logging. The empirical analysis confirmed the expectation of significantly improved time to repair from both system and media failures. This chapter also provided an in-depth description of the partitioned log index, a data structure which was initially envisioned for log archiving in support of media recovery, but turned out to be the essential building block of the novel approaches proposed in the remainder of the thesis.

Chapter 4 considered the problem of efficient update propagation under high transaction throughput. In the first part of this chapter, algorithms for asynchronous page cleaning were presented; this is an important component of a database system architecture, and yet recent research has not considered it in detail. The evaluation of page cleaning strategies shows that algorithms that make efficient use of main memory, while effectively propagating changes to the materialized database in a way that does not disturb transaction execution and guarantees ACID properties, is a major design and implementation challenge. As an alternative, novel log-based techniques were introduced and evaluated. While the empirical results showed a significant improvement in transaction throughput and propagation effectiveness, perhaps the most valuable contribution is the novel architecture that completely decouples persistence concerns from buffer management. This new decoupled architecture opens the possibility for many new ideas—one of which was proposed in the last part of this thesis.

Chapter 5 introduced the FineLine log-structured design for transactional storage and recovery. Its key design point is the use of the log as the only form of persistent storage, thus completely eliminating the materialized database. The main insight that led into this novel approach was the realization that a log data structure optimized for both writes and reads—i.e., the partitioned log index—makes actual database storage less relevant. Elements of this general idea were exploited in the log-based propagation strategy coupled with the write elision technique, but eliminating database storage essentially cuts many of the problems addressed in this thesis at their root, providing a simpler, more elegant, and generalized solution.

As it is perhaps expected from any research work, this thesis poses more questions than it answers; thus, many opportunities exist for future work. In the context of instant recovery, this thesis only implemented and evaluated its major components: instant restart and instant restore. The building blocks of these two techniques, however, enable a wide range of techniques for reliable and robust data management in general, perhaps most promisingly in the field of distributed databases with fast failover. An important step forward in that direction would therefore be the continued implementation and evaluation of instant recovery techniques in that context.

In the context of propagation strategies, this thesis only scratched the surface of the problem. Modern design proposals for in-memory database systems usually do not take this problem seriously enough, and the focus lies primarily on transaction performance. In a real-world deployment,

however, being able to efficiently propagate data to persistent storage while providing fast recovery, without giving up on any reliability feature such as media recovery, will become an increasingly pressing challenge. This is especially valid in the present time, when in-memory databases are quickly making their way into largely-deployed commercial products.

Lastly, the FineLine architecture still has a long road ahead. Implementing FineLine would by itself constitute a major science and engineering effort, which unfortunately could not be tackled in this thesis, especially considering the extent of the work on instant recovery and propagation strategies. Nevertheless, this thesis presented the initial idea and discussed possible approaches for the implementation of the architecture, including the reuse of many elements of the prototype used in the other chapters. The substantial improvements to in-memory transaction processing performance promised by the technique, as well as the potential to provide ACID-compliant persistence and instant recovery to arbitrary, memory-optimized data structures, are still left to be explored.

REFERENCES

[APD15] Joy Arulraj, Andrew Pavlo, and Subramanya Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proc. SIGMOD*, pages 707–722, 2015.

[APP16] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *PVLDB*, 10(4):337–348, 2016.

[BG88] Dina Bitton and Jim Gray. Disk Shadowing. In *Proc. VLDB*, pages 331–338, 1988.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BM04] Sorav Bansal and Dharmendra S. Modha. CAR: clock with adaptive replacement. In *Proc. FAST*, pages 187–200, 2004.

[BSR+06] Mary Baker, Mehul A. Shah, David S. H. Rosenthal, Mema Roussopoulos, Petros Maniatis, Thomas J. Giuli, and Prashanth P. Bungale. A fresh look at the reliability of long-term digital storage. In *Proc. EuroSys*, pages 221–234, 2006.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proc. SIGMOD*, pages 383–394, 1994.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[Chu17] Howard Chu. LMDB: Lightning Memory-mapped Database. https://symas.com/lightning-memory-mapped-database/, 2017. Accessed: 2017-05-05.

[CLG+94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, 1994.

[CMB+10] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. SSD bufferpool extensions for database systems. *PVLDB*, 3(2):1435–1446, 2010.

[Com79] Douglas Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[CSS+11] Tuan Cao, Marcos Antonio Vaz Salles, Benjamin Sowell, Yao Yue, Alan J. Demers, Johannes Gehrke, and Walker M. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proc. SIGMOD*, pages 265–276, 2011.

[DAI17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proc. SIGMOD*, pages 79–94, 2017.

[Den83] Peter J. Denning. The working set model for program behaviour (reprint). *Commun. ACM*, 26(1):43–48, 1983.

[DKO+84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *Proc. SIGMOD*, pages 1–8, 1984.

[DPT+13] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.

[DZP+11] Jaeyoung Do, Donghui Zhang, Jignesh M. Patel, David J. DeWitt, Jeffrey F. Naughton, and Alan Halverson. Turbocharging DBMS buffer pool using SSDs. In *Proc. SIGMOD*, pages 1113–1124, 2011.

[EB84] Klaus Elhardt and Rudolf Bayer. A database cache for high performance and fast restart in database systems. *ACM Trans. Database Syst.*, 9(4):503–525, 1984.

[EH84] Wolfgang Effelsberg and Theo Härder. Principles of Database Buffer Management. *ACM Trans. Database Syst.*, 9(4):560–595, 1984.

[ELL14] Ahmed Eldawy, Justin J. Levandoski, and Per-Åke Larson. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *PVLDB*, 7(11):931–942, 2014.

[Fou] Standard C++ Foundation. The Standard C++ FAQ. Available at: https://isocpp.org/wiki/faq/cpp11. Accessed: 2017-06-26.

[GG97] Jim Gray and Goetz Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record*, 26(4):63–68, 1997.

[GGS16] Goetz Graefe, Wey Guy, and Caetano 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.

[GHI+12] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi A. Kuno, and Stefan Manegold. Concurrency control for adaptive indexing. *PVLDB*, 5(7):656–667, 2012.

[GK85] Dieter Gawlick and David Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Eng. Bull.*, 8(2):3–10, 1985.

[GK10] Goetz Graefe and Harumi A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proc. EDBT*, pages 371–381, 2010.

[GK12] Goetz Graefe and Harumi A. Kuno. Definition, Detection, and Recovery of Single-Page Failures, a Fourth Class of Database Failures. *PVLDB*, 5(7):646–655, 2012.

[GKK12] Goetz Graefe, Hideaki Kimura, and Harumi A. Kuno. Foster b-trees. *ACM Trans. Database Syst.*, 37(3):17, 2012.

[GKS12a] Goetz Graefe, Harumi Kuno, and Bernhard Seeger. Self-diagnosing and self-healing indexes. In *Proc. DBTest*, pages 1–8. ACM, 2012.

[GKS12b] Goetz Graefe, Harumi A. Kuno, and Bernhard Seeger. Self-diagnosing and self-healing indexes. In *Proc. DBTest*, page 8, 2012.

[GLIa] GLIBC. The GNU C Library Reference Manual: Memory-mapped I/O. Available at: https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.htm. Accessed: 2017-05-31.

[GLIb] GLIBC. The GNU C Library Reference Manual: Renaming Files. Available at: http://www.gnu.org/software/libc/manual/html_node/Renaming-Files.html. Accessed: 2014-10-06.

[GLK+13] Goetz Graefe, Mark Lillibridge, Harumi A. Kuno, Joseph Tucek, and Alistair C. Veitch. Controlled lock violation. In *Proc. SIGMOD*, pages 85–96, 2013.

[GMB+81] Jim Gray, Paul R. McJones, Mike W. Blasgen, Bruce G. Lindsay, Raymond A. Lorie, Thomas G. Price, Gianfranco R. Putzolu, and Irving L. Traiger. The Recovery Manager of the System R Database Manager. *ACM Comput. Surv.*, 13(2):223–243, 1981.

[GP87] Jim Gray and Gianfranco 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.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra78] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, 1978.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12, 1986.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[Gra03a] Goetz Graefe. Sorting And Indexing With Partitioned B-Trees. In *CIDR*, 2003.

[Gra03b] Jim Gray. What next?: A dozen information-technology research goals. *J. ACM*, 50(1):41–57, 2003.

[Gra04] Goetz Graefe. Write-optimized b-trees. In *Proc. VLDB*, pages 672–683, 2004.

[Gra06a] Goetz Graefe. B-tree indexes for high update rates. *SIGMOD Record*, 35(1):39–44, 2006.

[Gra06b] Goetz Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3), 2006.

[Gra09a] Goetz Graefe. Fast loads and fast queries. In *Proc. DaWaK*, pages 111–124, 2009.

[Gra09b] Goetz Graefe. The five-minute rule 20 years later (and how flash memory changes the rules). *Commun. ACM*, 52(7):48–59, 2009.

[Gra11] Goetz Graefe. Modern B-Tree Techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[Gra12] Goetz Graefe. A survey of B-tree logging and recovery techniques. *ACM Trans. Database Syst.*, 37(1):1, 2012.

[GVK+14] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph Tucek, Mark Lillibridge, and Alistair C. Veitch. In-memory performance for big data. *PVLDB*, 8(1):37–48, 2014.

[HAMS08] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Proc. SIGMOD*, pages 981–992, 2008.

[Här05] Theo Härder. Architecture - the layer model and its evolution. *Datenbank-Spektrum*, 13:45–57, 2005.

[HR79] Theo Härder and Andreas Reuter. Optimization of Logging and Recovery in a Database System. In *IFIP TC-2 Working Conference on Data Base Architecture*, pages 139–156, 1979.

[HR83] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[HSH07] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. Architecture of a database system. *Foundations and Trends® in Databases*, 1(2):141–259, 2007.

[HSL+87] Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, Phil Garrett, and Andreas Reuter. Group commit timers and high volume transaction systems. In *Proc. HPTS*, pages 301–329, 1987.

[HSQ14] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware logging in transaction systems. *PVLDB*, 8(4):389–400, 2014.

[JCZ05] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the CLOCK replacement. In *Proc. USENIX*, pages 323–336, 2005.

[JPH+09] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. EDBT*, pages 24–35, 2009.

[JPS+12] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Scalability of write-ahead logging on multicore and multisocket hardware. *VLDB Journal*, 21(2):239–263, 2012.

[Kim15] Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proc. SIGMOD*, pages 691–706, 2015.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. ICDE*, pages 195–206, 2011.

[Knu98] Donald E Knuth. *The Art of Computer Programming, Volume 3: Sorting and searching*. Addison Wesley, 1998.

[KSHL12] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. *KISS-Tree*: smart latch-free in-memory indexing on modern architectures. In *Proc. DaMoN (SIGMOD Workshop)*, pages 16–23, 2012.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lar03] Per-Åke Larson. External sorting: Run formation revisited. *IEEE Trans. Knowl. Data Eng.*, 15(4):961–972, 2003.

[Lev91] Eliezer Levy. Incremental restart. In *Proc. ICDE*, pages 640–648, 1991.

[LKN13] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proc. ICDE*, pages 38–49, 2013.

[Lor77] Raymond A. Lorie. Physical integrity in a large segmented database. *ACM Trans. Database Syst.*, 2(1):91–104, 1977.

[LS90] Eliezer Levy and Abraham Silberschatz. Log-driven backups: A recovery scheme for large memory database systems. In *Proc. Jerusalem Conference on Information Technology*, pages 99–109, 1990.

[LS92] Eliezer Levy and Abraham Silberschatz. Incremental recovery in main memory database systems. *IEEE Trans. Knowl. Data Eng.*, 4(6):529–540, 1992.

[LSG+79] B.G. Lindsay, P.G. Selinger, C. Galtieri, J.N. Gray, R.A. Lorie, T.G. Price, F. Putzolu, and B.W. Wade. Notes on distributed databases. *IBM Res. Rep. RJ 2571*, 1979.

[LSH16] Lucas Lersch, Caetano Sauer, and Theo Härder. Decoupling persistence services from DBMS buffer management. In *Proc. GI-Workshop "Grundlagen von Datenbanken"*, pages 80–85, 2016.

[LSKN16] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proc. DaMoN (SIGMOD Workshops)*, pages 3:1–3:8, 2016.

[M+04] Cary Millsap et al. *Oracle Insights: Tales of the Oak Table*. Apress, 2004.

[MAZ+16] Lin Ma, Joy Arulraj, Sam Zhao, Andrew Pavlo, Subramanya R. Dulloor, Michael J. Giardino, Jeff Parkhurst, Jason L. Gardner, Kshitij Doshi, and Stanley B. Zdonik. Larger-than-memory data management on modern storage hardware for in-memory OLTP database systems. In *Proc. DaMoN Workshop*, pages 9:1–9:7, 2016.

[MHL+92] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter 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.

[Mic17a] Microsoft. Automatic Page Repair (Availability Groups: Database Mirroring). https://docs.microsoft.com/en-us/sql/sql-server/failover-clusters/automatic-page-repair-availability-groups-database-mirroring, 2017. Accessed: 2017-05-18.

[Mic17b] Microsoft. Transaction Log Truncation in SQL Server. https://technet.microsoft.com/en-us/library/ms189085.aspx, 2017. Accessed: 2017-05-09.

[MKM12] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proc. EuroSys*, pages 183–196, 2012.

[ML92] C. Mohan and Frank E. Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In *Proc. SIGMOD*, pages 371–380, 1992.

[MM03] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proc. FAST*, 2003.

[MN93] C. Mohan and Inderpal Narang. An Efficient and Flexible Method for Archiving a Data Base. *SIGMOD Rec.*, 22(2):139–146, June 1993.

[Moh90a] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *Proc. VLDB*, pages 392–405, 1990.

[Moh90b] C. Mohan. Commit_lsn: A novel and simple method for reducing locking and latching in transaction processing systems. In *Proc. VLDB*, pages 406–418, 1990.

[Moh93] C. Mohan. A cost-effective method for providing improved data availability during DBMS restart recovery after a failure. In *Proc. VLDB*, pages 368–379, 1993.

[Moh95] C. Mohan. Disk read-write optimizations and data integrity in transaction systems using write-ahead logging. In *Proc. ICDE*, pages 324–331, 1995.

[MWMS14] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory OLTP recovery. In *Proc. ICDE*, pages 604–615, 2014.

[OCGO96] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[OLN+16] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proc. SIGMOD*, pages 371–386, 2016.

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. SIGMOD*, pages 297–306, 1993.

[P+02] David Patterson et al. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical report, UCB//CSD-02-1175, UC Berkeley, 2002.

[PDTP16] Kwanghyun Park, Jaeyoung Do, Nikhil Teletia, and Jignesh M. Patel. Aggressive buffer pool warm-up after restart in SQL Server. In *Proc. ICDE*, pages 31–38, 2016.

[Pos] PostgreSQL. PostgreSQL 9.6 documentation, Section 19.4, Resource Consumption. Available at: https://www.postgresql.org/docs/9.6/static/runtime-config-resource.html. Accessed: 2017-06-16.

[PWGB13] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage management in the NVRAM era. *PVLDB*, 7(2):121–132, 2013.

[RBM13] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: the linux b-tree filesystem. *ACM Trans. Storage*, 9(3):9:1–9:32, 2013.

[RDAT16a] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proc. SIGMOD*, pages 1539–1551, 2016.

[RDAT16b] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proc. SIGMOD*, pages 1539–1551, 2016.

[Reu80] Andreas Reuter. A fast transaction-oriented logging scheme for UNDO recovery. *IEEE Trans. Software Eng.*, 6(4):348–356, 1980.

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[Rod08] Ohad Rodeh. B-trees, shadowing, and clones. *TOS*, 3(4), 2008.

[SA13] Radu Stoica and Anastasia Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. In *Proc. DaMoN Workshop*, page 7, 2013.

[SGH14] Caetano Sauer, Goetz Graefe, and Theo Härder. An empirical analysis of database recovery costs. In *RDSS (SIGMOD Workshops), Snowbird, UT, USA*, 2014.

[SGH15] Caetano Sauer, Goetz Graefe, and Theo Härder. Single-pass restore after a media failure. In *Proc. BTW, LNI 241*, pages 217–236, 2015.

[SGH17] Caetano Sauer, Goetz Graefe, and Theo Härder. Instant restore after a media failure. *CoRR*, abs/1702.08042, 2017.

[SH14] Caetano Sauer and Theo Härder. A novel recovery mechanism enabling fine-granularity locking and fast, REDO-only recovery. *CoRR*, abs/1409.3682, 2014.

[SK07] Jayson Speer and Markus Kirchberg. C-ARIES: A multi-threaded version of the ARIES recovery algorithm. In *Proc. DEXA*, pages 319–328, 2007.

[SL76] Dennis G. Severance and Guy M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst.*, 1(3):256–267, 1976.

[SLHG16] Caetano Sauer, Lucas Lersch, Theo Härder, and Goetz Graefe. Update propagation strategies for high-performance OLTP. In *Proc. ADBIS*, 2016.

[SR12] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. In *Proc. SIGMOD*, pages 217–228, 2012.

[Sto87] Michael Stonebraker. The design of the POSTGRES storage system. In *Proc. VLDB*, pages 289–300, 1987.

[Sto16] Michael Stonebraker. The land sharks are on the squawk box. *Commun. ACM*, 59(2):74–83, 2016.

[TA10] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *PVLDB*, 3(1):70–80, 2010.

[Tar12] Vasily Tarasov. Linux Asynchronous I/O Explained. Available at: https://www.fsl.cs.sunysb.edu/~vass/linux-aio.txt, 2012. Accessed: 2017-06-05.

[TZK+13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proc. SIGOPS*, pages 18–32, 2013.

[VWA+12] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. Logbase: A scalable log-structured database system in the cloud. *PVLDB*, 5(10):1004–1015, 2012.

[Wei91] Gerhard Weikum. Principles and realization strategies of multilevel transaction management. *ACM Trans. Database Syst.*, 16(1):132–180, 1991.

[Wik17] Wikipedia. Logistic function. http://en.wikipedia.org/wiki/Logistic_function, 2017. Accessed: 2017-05-19.

[WJ14] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.

[WV02] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[YAC+16] Chang Yao, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, and Sai Wu. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *Proc. SIGMOD*, pages 1119–1134, 2016.

[YMUY10] Makoto Yui, Jun Miyazaki, Shunsuke Uemura, and Hayato Yamana. Nb-GCLOCK: A non-blocking buffer management based on the generalized CLOCK. In *Proc. ICDE*, pages 745–756, 2010.

[ZTKL14] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proc. USENIX*, pages 465–477, 2014.

ACKNOWLEDGMENTS

This thesis was the result of a long and strenuous journey, in which the following people had a crucial contribution: Theo Härder, for being a true *Doktorvater*, investing in me by bringing me from Brazil to an exchange program in his group back in 2008, providing the means to support my livelihood in all these years, and guiding me all the way to my PhD graduation; Goetz Graefe, for being my mentor, giving me an exciting and very industry-relevant research topic to work on, and deeply influencing the way I think about designing system software; Thomas Neumann, for agreeing to be part of my defense committee, teaching and influencing me in multiple ways, and investing in my future career; Stefan Deßloch, for supporting me through my studies, teaching activities, and various bureaucratic procedures with the University; Sebastian Michel, for many helpful conversations and direct support of my work; Heike Neu, for always helping the people in our group with kindness and dedication; my former and current colleagues, for sharing the PhD experience and offering support in many ways; my Brazilian friends of the DBIS group, for sharing the experience of living and working in the second floor of building 36, from the stressful deadlines to the small joys of coffee, lifting weights, philosophical insights, and immature jokes; and my loving wife Irina, for making all of this worthwhile and always being there for me.

ABOUT THE AUTHOR

Caetano Sauer was born on December 4, 1987, in Três Passos, a small city in the south of Brazil, close to the Argentinian border. The city's foundation and development, from the late 19th century onwards, has had significant influence of German immigrants from the Hunsrück region, which is not far from Kaiserslautern. After spending his childhood and teenage years in the central-western city of Cuiabá, he moved back to the south, to the state capital Porto Alegre, to begin his Computer Science studies in 2005. Three years later, in 2008, he came to Kaiserslautern to work as a research intern in the group of Prof. Theo Härder. What was initially planned as a one-year internship turned into a permanent relocation; he transferred his Bachelor studies to the TU Kaiserslautern and graduated in 2009, after which the Master's degree was obtained in 2012. Since then, he has been a PhD candidate under Prof. Härder, with close collaboration and mentorship from Dr. Goetz Graefe. The present thesis marks the end of this incredible journey.

Education

Jul 2012 – Aug 2017	PhD, Computer Science, TU Kaiserslautern
Apr 2010 – Jun 2012	M.Sc, Computer Science, TU Kaiserslautern
Mar 2009 – Apr 2010	B.Sc., Computer Science, TU Kaiserslautern
Mar 2005 – Dec 2007	B.Sc., Computer Science, UFRGS, Brazil (unfinished)

Work experience

since Oct 2017	Senior Software Engineer, Tableau Software, Munich, Germany
Jul 2012 – Sep 2017	Scientific Staff Member, Databases and Information Systems Group, TU Kaiserslautern, Germany
Feb 2008 – Jun 2012	Research Intern, Databases and Information Systems Group, TU Kaiserslautern, Germany
Mar 2007 – Dec 2007	Research Intern, Microsoft Innovation Center, UFRGS, Brazil