T. Grust et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2019),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2019 487

Modern techniques for transaction-oriented database recovery[1]

Caetano Sauer[2]

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 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, the ever-increasing capacity of magnetic disks, and, on the long term, the potential adoption of non-volatile memory. In our research, we evaluated a variety of new software techniques for efficient transaction-oriented database recovery, focusing on availability and architectural simplicity. The techniques presented here differ from most recent work in the field in which they aim to be *hardware-agnostic*, supporting different memory and storage configurations with the same software, as well as *fully functional* in comparison with traditional database systems, e.g., by supporting media recovery, index management, larger-than-memory datasets, and arbitrary access structures with structural modifications.

1 Motivation

Most existing approaches for database recovery based on write-ahead logging (WAL), including the widely used ARIES [Mo92] methods implemented in the vast majority of commercial database systems, suffer from two major problems. 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, a new technique called *instant recovery* enables access to individual data pages (or contiguous sets thereof) before the complete recovery process is finished. These techniques cover all classes of database failures, most notably system and media failures. By performing recovery actions on demand, system

[1] This short article summarizes a doctoral dissertation with the same title [Sa17]

[2] Tableau Software, csauer@tableau.com (Dissertation completed at TU Kaiserslautern)

doi:10.18420/btw2019-30

availability as observed by individual transactions can be effectively increased by up to two "nines" by means of simple and incremental software techniques.

Besides increasing availability by optimizing the recovery process, efficient *checkpointing* as well as *backup* techniques are required to maintain the persistent database in a fresh state and thus effectively reduce the amount of recovery work to be performed in case of a failure. Building upon a discussion of efficient checkpoint algorithms, as well as on ideas from instant recovery, this work 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.

Looking beyond traditional WAL architectures, these ideas are taken further with a novel design for database storage and recovery called *FineLine*. Its goal is to simplify the recovery and checkpointing processes by eliminating the persistent database, relying solely on the recovery log for data storage and retrieval. With FineLine, there is no duality between persisted database and log and, as a consequence, no algorithmic logic that is exclusive to recovery from failures; instead, recovery is embedded in the data access protocol, without distinction between normal and recovery processing modes. This results in a much simpler system architecture, which also decouples in-memory data structures from persistence concerns and reduces log volume, thus improving performance for memory-resident workloads. 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.

If a system is able to keep the amount of recovery work manageable and perform this recovery work not only while transactions are running but also prioritizing the needs of those transactions, then the challenge postulated by Jim Gray in his ACM 1998 Turing Award Lecture of a system "unavailable for less than one second per hundred years" [Gr03] gets one step closer to becoming a reality.

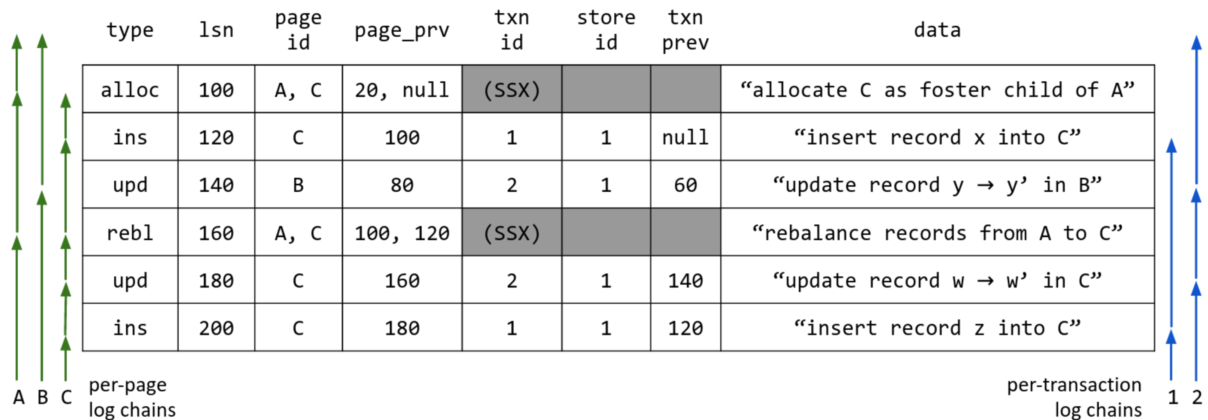
2 Key contributions

This section summarizes the contributions of the author's dissertation in five parts. Further details are provided in the cited references as well as in the published dissertation [Sa17].

2.1 Instant restart

Instant restart is a technique for recovery from system failures that builds upon single-page repair [GK12] 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. This requires exploiting the independence of recovery among objects that is inherent to physiological logging [Mo92] 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 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.



type	lsn	page id	page_prv	txn id	store id	txn prev	data
alloc	100	A, C	20, null	(SSX)			“allocate C as foster child of A”
ins	120	C	100	1	1	null	“insert record x into C”
upd	140	B	80	2	1	60	“update record y → y’ in B”
rebl	160	A, C	100, 120	(SSX)			“rebalance records from A to C”
upd	180	C	160	2	1	140	“update record w → w’ in C”
ins	200	C	180	1	1	120	“insert record z into C”

Fig. 1: Example of log records with per-page and per-transaction chains

Log-record chains are a fundamental technique to enable on-demand restart recovery. They are illustrated in Fig. 1, which shows an example of log snippet in which two transactions, 1 and 2, make modifications to three pages, A, B, and C. In this log, two independent system transactions, identified with `SSX`, are performing a split on page A of a B-tree data structure in two steps: first, a new page C is allocated, and then records are moved from A to C in a *rebalance* operation. On the right side of the diagram, two in-flight user transactions are shown with their per-transaction log chains; these are used for undo recovery and are also present in the original ARIES design [Mo92]. Note that user transaction 1 is allowed to

insert a record in page C while the page split is still happening—this illustrates the utility of system transactions as well as the fundamental distinction between database contents and their representation [Gr12].

The left side of the diagram shows per-page log chains for pages A, B, and C. Each log record points to the last log record that modified that same page. This chain can be easily maintained by saving the page LSN value into the log record before applying its corresponding update and setting the new page LSN to its own LSN value [GGS14].

To understand instant restart by example, consider that a system failure occurred and the log of Figure 1 is all the log that is found after restart and that none of its updates has been propagated to the database. In that case, the log analysis phase determines that transactions 1 and 2 need to be undone and pages A, B, and C need to be redone. The exclusive locks held by transactions 1 and 2 (i.e., the loser transactions) before the failure are also reacquired during log analysis. This information can be kept in the lock manager and in the buffer manager, respectively, as special entries that also mark the head of each log chain. Once log analysis is finished, and, most importantly, *before any undo or redo actions take place*, the system is made available for new transactions.

Post-failure transactions have unrestricted access to the database: if they fix a page in the buffer pool that needs redo recovery, the old image is fetched and redo recovery is performed, on that page only, by following the per-page log chain until the page LSN value and reapplying the log records in reverse order (i.e., using a stack). Following the *repeating history* paradigm, updates of loser transactions are also redone [Mo92].

Once a page is fixed and all its updates redone, it may still contain updates of loser transactions, which have to be undone. This is achieved by aborting a loser transaction on demand as soon as a post-failure transaction requests a lock on a record or key value that is currently held by it (i.e., a lock reacquired during log analysis).

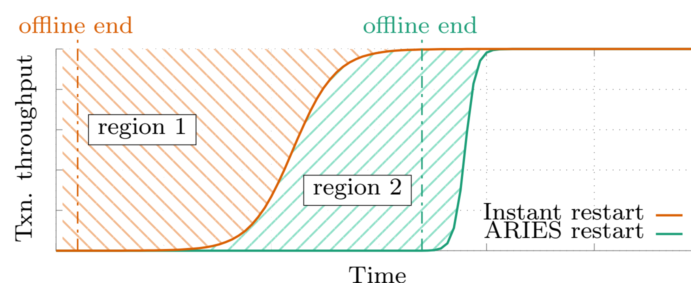


Fig. 2: Logistic function patterns of instant restart and ARIES restart

The performance of instant restart depends on factors such as storage hardware characteristics, transaction volume, and workload behavior. These were evaluated in detail in the dissertation [Sa17], but the general behavior expected during restart is that of a *logistic function* depicted in Fig. 2, where time is shown in the x-axis and transaction throughput in the y-axis. The chart shows the behavior of instant restart in comparison with ARIES restart as two logistic curves and their respective *offline phases*, i.e., the points for which transaction

throughput is zero. With instant restart, the offline phase finishes much earlier than ARIES, and recovery is performed as a side-effect of the workload access pattern; thus, throughput is increased more gradually as the buffer pool is warmed up. The net result is that the number of transactions missed due to a failure—the shaded area labeled “region 1”–is significantly lower. Once again, we refer to the dissertation for concrete numbers under different experiment configurations [Sa17].

2.2 Single-pass restore

The benefit of on-demand recovery can also be achieved for media failures, but since the amount of log that must be accessed during media recovery is likely to be many orders of magnitude larger than during restart recovery, we first address the issue of media recovery efficiency with a technique called *single-pass restore* [SGH15].

Single-pass restore maintains the log archive in a partially sorted organization, by introducing a run generation phase to the log archiving process. Within each run, log records are sorted primarily by page identifier rather than by LSN, so that a merge of log archive partitions and an old database backup is able to produce an up-to-date database using a single sequential pass, rather than a sequential pass of the log with random accesses on a database backup and its replacement device. The process is illustrated in Fig. 3, and more details are available in the original publication [SGH15].

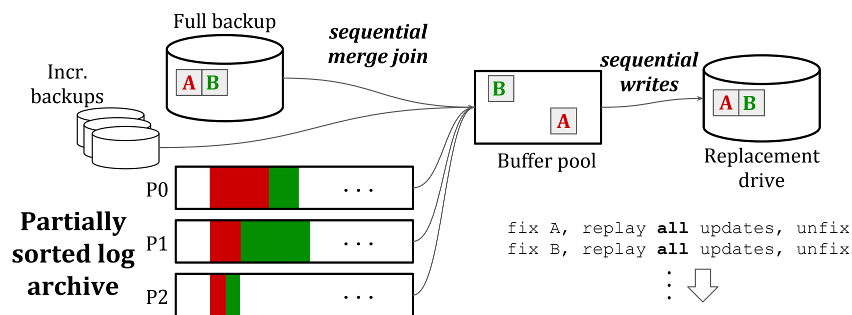


Fig. 3: Single-pass restore

In our measurements, run generation during log archiving adds less than 3% overhead to regular forward processing [SGH15], and the gained benefit is that full recovery of an up-to-date database takes the same amount of time as copying an old backup image in traditional methods. Single-pass restore also makes the management of backups much simpler, renders incremental backups obsolete, and enables system administrators to substantially reduce the frequency of full backups without compromising availability [GGS14].

2.3 Instant restore

Instant restore [SGH17] builds upon the partially sorted log archive of single-pass restore by adding an index to each sorted partition. This enables the retrieval of log records of a given database page or—more appropriately to exploit sequential access speeds—a contiguous set of database pages, which we call a *segment*. Such an *indexed log archive* enables on-demand restoration of database segments in the same fashion that individual pages are redone in instant restart; the only difference is that log records are retrieved from an index rather than by following a chain of log records.

Despite sharing the same general principle of on-demand recovery, instant restart after a system failure and instant restore after a media failure are quite different in their causes, effects, and recovery measures. When a media failure is detected, the system process is not affected, and thus its main-memory state—including buffer pool, transaction and lock managers, etc.—is not lost. This is unlike a system failure, or *crash*, in which all volatile state is lost. This means that user transactions may continue execution after a media failure, provided that they only access data that is either in the buffer pool or on other (still healthy) persistent devices. A transaction that accesses data on a failed device is normally aborted, but with instant restore, it can actually trigger the restoration of the segments that contain the accessed pages on demand. Thus, a moderate delay is added to the transaction's response time (in addition to the page miss that would have occurred even without the media failure) rather than aborting it.

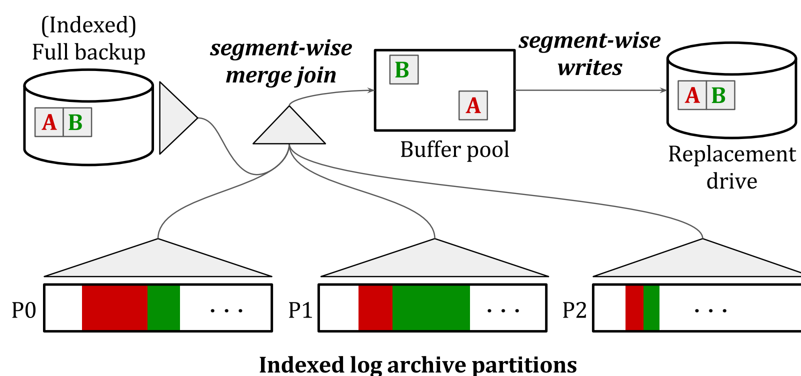


Fig. 4: Illustration of instant restore

Restoration of a failed device from a full backup and an indexed log archive is illustrated in Fig. 4, which shows an example segment consisting of just two pages, A and B. Thanks to the grouping of pages into segments and the partially-sorted order of log archive partitions, the access pattern is mostly sequential, and thus recovery is as efficient as single-pass restore. Thanks to the index added on top of the sorted log partitions, segments which are needed most immediately by applications are prioritized, while the remaining "cold" segments can be restored in the background.

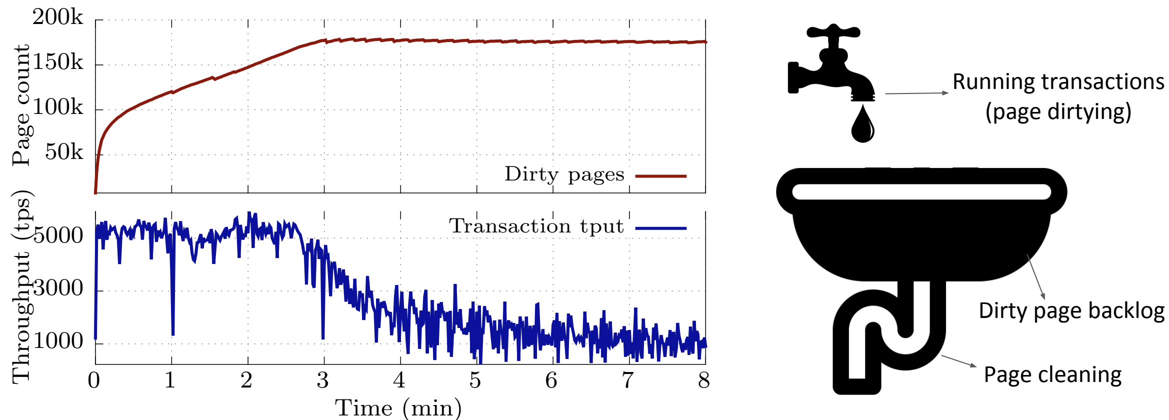


Fig. 5: Dirty page backlog and its implication on system performance

2.4 Decoupled persistence

Our work on propagation strategies [Sa16] emphasizes the problem of efficiently propagating updates from the buffer pool (i.e., cached dirty data in volatile storage) to persistent storage. It highlights the need for a well-balanced architecture in which the aggregate bandwidth of propagation matches the update rate of the transaction workload—in other words, in which the *cleaning speed* matches the *dirtying speed*. This problem is especially relevant for write-heavy workloads (such as TPC-C) running on memory-abundant systems. In such cases, inefficient propagation may lead to a disk bottleneck on transaction throughput, even though an application's working set fits entirely in main memory, as illustrated in Fig. 5 (more details of this experiment in the original publication [Sa16]).

These observations lead to the insight that update propagation for memory-intensive and write-heavy applications must happen under control of a dedicated background service called the *page cleaner* [Sa17] rather than on a page-by-page basis and triggered exclusively by a page replacement strategy, which is the traditional approach. Such a cleaner service must decide, from a set of dirty candidate pages, which pages to write back to disk at a particular point in time. This choice depends on a model that takes into account the benefit of writing a particular page and the cost of doing so for a given storage hardware configuration. In addition to these observations, our work proposes a log-based model in which updates are propagated not by writing cached pages directly, but by replaying log records from the partitioned index presented earlier for instant restore [Sa16]. This idea, when taken to the extreme, leads to the system design presented in the next section.

2.5 FineLine

FineLine [SGH18] is a novel system design that stores all data in the log, thus departing from a traditional write-ahead logging architecture, in which persistent data lives both in the log and in a database file. Unlike a traditional, time-ordered log, the FineLine log is

actually a partitioned index on database page identifiers, as proposed for instant restore. Propagation of updates from volatile caches to persistent storage happens solely via logging, which contains only redo information thanks to its *no-steal* approach [HR83]. In order to fetch a page from persistent storage, its state is simply reconstructed from its pertaining log records, as illustrated in Fig. 6.

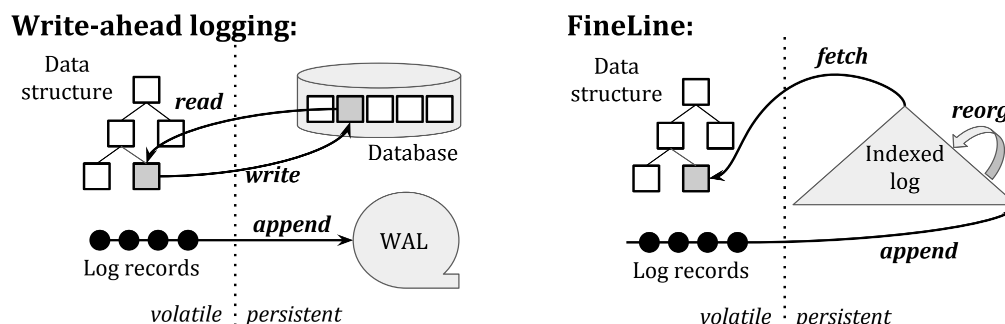


Fig. 6: Propagation of updates in WAL (left) vs. FineLine (right)

The partitions in the log are periodically merged in order to deliver acceptable read performance, similar to log-structured merge trees [ON96]. The key difference is that FineLine relies on physiological logging, thus indexing log records by their page identifier rather than by key values in the application domain. As such, it can be seen as a persistence module that supports transactional durability to arbitrary in-memory data structures. Unlike log-structured merge trees, the log index itself provides durability and there is no separate write-ahead log. By relying on physiological logging, FineLine also inherits all the benefits of ARIES and instant recovery, including on-demand system and media recovery, support for system transactions for space management and structural operations, secondary indexes, buffer management, and orthogonality to isolation mechanisms (i.e., two-phase locking, optimistic validation, multi-versioning, etc. are all supported). Lastly, compared to a Shore-MT-based prototype that implements ARIES-based logging and recovery, FineLine improves performance of a memory-resident workload by about 2×, requires less code, and is more modular [SGH18].

3 Lessons learned and outlook

In this dissertation, we attempted to improve the availability and efficiency of logging and recovery algorithms for transaction-oriented database systems. The following paragraphs summarize some guiding principles that we followed in this work and are likely also relevant for future work.

Database systems are effective and versatile tools that build the backbone of some of the most critical and useful applications in the world. Given this key position, database systems are also extremely performance-critical, and thus a large share of research efforts in the field are invested towards making databases as fast as possible. However, in many cases, optimizing for performance conflicts with the goals of versatility and data independence.

As in many endeavors in science and engineering, the challenge therefore lies in finding the correct compromise.

Over the last decade, the advent of cheap main memory and multi-core CPUs pushed database researchers to reconsider their architectures. Perhaps most influential in this effort was the work of Stonebraker et al. on H-Store [St07]. This system achieves orders-of-magnitude improvement in transactional performance by completely eliminating components such as logging and recovery, locking, and buffer management. While this work served as an important "wake-up call" that deeply influenced many of the systems designed in the last decade, the rather extreme approach of trimming off as much functionality as possible for the goal of performance is likely too restrictive for database systems in general. Since H-Store was proposed, research papers that followed gradually reintroduced functionality such as concurrency control, recovery, and buffer management. In a way, the "complete rewrite" was extremely useful as a reconsideration of software architectures in the light of modern hardware, but the end result (if there is one) might look much more similar to a traditional architecture than expected.

In the context of transaction recovery, our work advocates for the retainment of certain fundamental design principles, such as physiological logging and buffer management. These are crucial to provide functionality such as media recovery (by means other than replication and failover), native indexing support, access-path (i.e., data-structure) independence, system transactions, and independence of concurrency control mechanisms (e.g., fine-grained two-phase locking). Our benchmark has been to support all the functionality that ARIES [Mo92] supports—if our work can be as applicable as the most ubiquitous recovery mechanism in the history of database systems, then we have indeed succeeded across at least one dimension.

The "main-memory revolution" of the last decade has been, in a perhaps quite ironic way, accompanied by a revolution in storage hardware architectures. In fact, this "storage revolution" is still going on and will likely continue until non-volatile memory (NVM) is established in the market and its development somehow stabilizes. Flash memory and fast solid-state drives already changed many assumptions of the storage hierarchy, but upcoming NVM storage has the potential to beget another "complete rewrite". While preliminary proposals for NVM-based transactional systems are promising, many of them wage on NVM "taking over" the storage hierarchy, while, in reality, the storage landscape is likely to remain heterogeneous.

Given the uncertainty about how a future storage hierarchy might look like, the unlikelihood of it being NVM-only, and the lessons learned from the main-memory revolution about versatility and applicability, our work focused on hardware-agnostic software techniques, which have the potential to be optimized for NVM rather than being designed exclusively for it. We believe this is an important principle to follow if our research community should invent a "new ARIES" that will become standard in database systems to come.

Acknowledgments: My “doctoral father” Theo Härder took me under his wing as an undergraduate research assistant from Brazil, supported my work all the way through a PhD, and continues to do so as I moved into industry. I will be forever grateful for that. Goetz Graefe was not only an invaluable source of knowledge with his great contributions to the field but also co-advisor of my thesis and a mentor who was always available and inspired me with his approach to building complex systems using simple software techniques. Thomas Neumann kindly agreed to participate in my dissertation committee and is also a source of inspiration for his great work, thanks to which I now have an exciting and challenging job.

References

[GGS14] Graefe, G.; Guy, W.; Sauer, C.: Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, and Media Restore. Morgan & Claypool, 2014.

[GK12] Graefe, G.; Kuno, H. A.: Definition, Detection, and Recovery of Single-Page Failures, a Fourth Class of Database Failures. PVLDB 5/7, pp. 646–655, 2012.

[Gr03] Gray, J.: What next?: A dozen information-technology research goals. J. ACM 50/1, pp. 41–57, 2003.

[Gr12] Graefe, G.: A survey of B-tree logging and recovery techniques. ACM Trans. Database Syst. 37/1, p. 1, 2012.

[HR83] Härder, T.; Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. 15/4, pp. 287–317, 1983.

[Mo92] Mohan, C.; Haderle, D.; Lindsay, B.; Pirahesh, H.; Schwarz, P.: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS 17/1, pp. 94–162, 1992.

[ON96] O’Neil, P. E.; Cheng, E.; Gawlick, D.; O’Neil, E. J.: The Log-Structured Merge-Tree (LSM-Tree). Acta Inf. 33/4, pp. 351–385, 1996.

[Sa16] Sauer, C.; Lersch, L.; Härder, T.; Graefe, G.: Update Propagation Strategies for High-Performance OLTP. In: Proc. ADBIS, LNCS 9809, pp. 152–165. 2016.

[Sa17] Sauer, C.: Modern techniques for transaction-oriented database recovery, PhD thesis, Dr. Hut Verlag, pp. 1–141: TU Kaiserslautern, Germany, 2017.

[SGH15] Sauer, C.; Graefe, G.; Härder, T.: Single-pass restore after a media failure. In: Proc. BTW, LNI 241, pp. 217–236. 2015.

[SGH17] Sauer, C.; Graefe, G.; Härder, T.: Instant Restore After a Media Failure. In: Proc. ADBIS, LNCS 10509, pp. 311–325. 2017.

[SGH18] Sauer, C.; Graefe, G.; Härder, T.: FineLine: log-structured transactional storage and recovery. PVLDB 11/13, pp. 2249–2262, 2018.

[St07] Stonebraker, M.; Madden, S.; Abadi, D. J.; Harizopoulos, S.; Hachem, N.; Helland, P.: The End of an Architectural Era (It’s Time for a Complete Rewrite). In: Proc. VLDB, pp. 1150–1160. 2007.