Decoupling Persistence Services from DBMS Buffer Management

Lucas Lersch[*]
TU Kaiserslautern
Germany
lucas.lersch@sap.com

Caetano Sauer
TU Kaiserslautern
Germany
csauer@cs.uni-kl.de

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

ABSTRACT

Advances in DRAM technology and, in turn, substantial cost reduction for volatile memory in recent years require an evolution of database system architectures to take full benefit of large buffer pools. Having huge memory sizes, an up-to-date version of database pages on stable storage is more than ever necessary to support fast and effective crash recovery.

In this contribution, we consider important components of a traditional DBMS architecture and related opportunities for optimization in the context of persistence and system recovery. We implemented and evaluated novel checkpointing and page cleaning algorithms which are based on log information rather than on data collected from critical in-memory data structures such as buffer pool and transaction manager. Decoupling such persistence-related components from in-memory processing enables a simpler and more modular DBMS architecture as well as less interference with these components in critical in-memory data structures.

Keywords

database architecture, transaction processing, recovery

1. INTRODUCTION

The architecture of traditional database systems evolved in a time when the available amount of main memory was relatively small compared to the size of typical application working sets. To achieve satisfactory transaction performance, these systems had to be heavily optimized for the underlying hardware, i. e., the design had to consider frequent operations to persistent storage, taking into account high latencies of both commit operations and data accesses. Due to current advances in hardware technology, however, an adaptation of existing DBMS architectures is required to unleash their inherent performance potential [11].

Compared to the situation in the 1970s and 1980s, one of the most important hardware-related changes is the dramatic cost reduction of volatile memory. Nowadays, it is realistic to consider a scenario where the buffer pool can accommodate most, if not all, data pages and, consequently, there are fewer page reads and writes from/to persistent storage. Furthermore, the commit latency is drastically reduced, thanks to modern solid state drives which offer a much higher number of I/O operations per second when compared to hard-disk drives. In this contribution, we reconsider the existing components of a traditional database architecture and opportunities for optimization in the context of persistence and system recovery.

We assume that even with large amounts of memory and transaction processing hardly requiring I/O operations, it is still desirable to keep an up-to-date version of database pages in persistent storage to guarantee efficient recovery in case of a system failure. Most modern database systems rely on write-ahead logging techniques and implement the ARIES [6] algorithm for system recovery. In such environments, regular checkpoints are taken to improve recovery performance. Checkpoints serve as the starting point for log analysis, the first phase of crash recovery. The more recent a checkpoint was taken, the less time log analysis takes to complete in case of a failure.

In addition to checkpoints, it is also common for database systems to periodically clean dirty pages (pages with committed updates so far not propagated) by flushing them from the in-memory buffer pool to persistent storage. Consequently, since the REDO phase (after a crash) would require random reads, its cost is reduced by maintaining only a low count of dirty pages [7]. In this work, we refer to both checkpointing and page cleaning in a general way as *propagation services*, in the sense that they propagate information from the main memory to persistent storage.

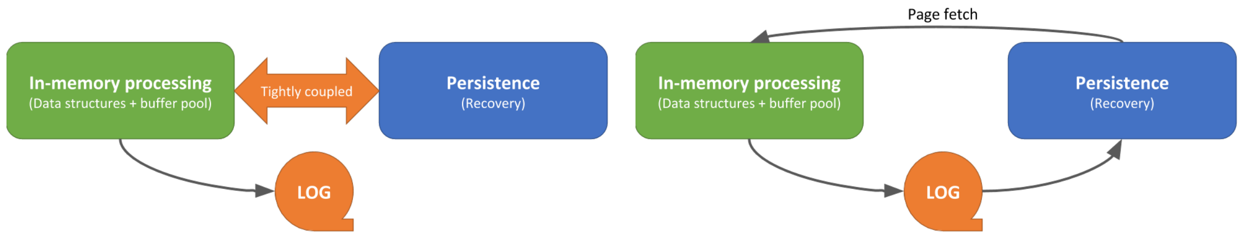
The problem is that such propagation services interfere with and might disturb in-memory transaction processing, since they must inspect data structures and consequently acquire and release latches on them. This is especially true for scenarios of large buffer pools capable of holding the entire application working set, since disk I/O is less likely to be the bottleneck. Furthermore, decoupling unnecessarily coupled components in a system is not only a matter of performance, but it also offers a cleaner, more modular system architecture and reduces code complexity.

The first contribution of the proposed architecture is to enable checkpoints to gather all required information solely by inspecting the log. As a main advantage, it is not only simpler by making use of the same existing logic of log analysis during recovery, but it also implies less interference

[*] Currently with TU Dresden & SAP SE.

28th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 24.05.2016 - 27.05.2016, Nörten-Hardenberg, Germany.

Copyright is held by the author/owner(s).



(a) Tightly coupled persistence

(b) Decoupled persistence

Figure 1: Differences between traditional and proposed architecture.

with in-memory processing, as mentioned above. The second contribution is a decoupled implementation of a page cleaner, which also makes use of log data instead of in-memory information. Besides also reducing the interference in the buffer pool, it can be combined with single-page recovery techniques [3] that enable a set of interesting features for the recovery component of a database system, as discussed later. Figure 1 illustrates differences between the traditional design and the proposed decoupled design.

2. BACKGROUND

2.1 System Recovery

2.1.1 ARIES Restart

To offer transaction atomicity and durability guarantees, i.e., the "A" and "D" of ACID, most modern database systems implement a write-ahead logging mechanism (WAL). The ARIES algorithm [6] embodies an efficient way to enable WAL-based system recovery. In case of a system crash, the recovery goal is to re-establish the most recent transaction-consistent database state.

The system recovery algorithm operates in three different phases: log analysis, REDO, and UNDO. Starting from the last checkpoint, log analysis scans the log to inspect all log records to determine which pages were dirty and which transactions were active at the time of failure. After having analyzed the log, the REDO phase scans the relevant part of the log and replays all log records that refer to updates of dirty pages. Finally, the UNDO phase rolls back transactions that were active at the time of failure.

As observed in a previous analysis of typical OLTP workloads [7], UNDO is usually very short, and the determinant factor for recovery cost is the number of dirty pages that require log replay in the REDO phase. Therefore, it is crucial to clean pages proactively as frequently and efficiently as possible. Furthermore, it is desirable to take frequent checkpoints to reduce the time required for log analysis. In a traditional design with tightly-coupled propagation services, however, an increased frequency of page cleaning and checkpoints inevitably hurts performance, as shown in Section 4.

2.1.2 Instant Restart

An alternative design for system recovery is the instant restart mechanism [2], which opens the system for new transactions immediately after the log analysis phase, performing the required REDO and UNDO actions on demand. The design is implemented with moderate incremental changes to the traditional ARIES algorithm. On-demand REDO of dirty pages is performed page by page instead of using a log scan. To that end, each log record must include a pointer (i.e., the LSN) of the last log record affecting a specific page. Such a pointer is simply copied from the page LSN field prior to generating a log record.

For on-demand UNDO, loser transactions are rolled back concurrently to REDO and newly running transactions. This rollback is triggered by lock conflicts between new and loser transactions. Therefore, log analysis requires a lock reacquisition for loser transactions, which implies that currently-held locks must also be included in the information collected for checkpoints – this is in contrast to the ARIES method which re-acquires such locks in the REDO phase. For a traditional checkpoint procedure inspecting in-memory data structures, this effort causes additional interference, which is eliminated in our decoupled design.

For further details, we refer to an extensive publication on instant recovery techniques [2].

2.2 Checkpoints

In a general way, a checkpoint could be defined as a relatively recent persistent copy of any information that serves as a basis for restart and enables a faster recovery process. The more recent a checkpoint was taken w.r.t. the time of the system failure, the less work is required for recovering the system. Thus, it is a good practice to take checkpoints regularly. To be more precise in the definition of a checkpoint, we follow the definition of *fuzzy checkpoints* as presented in [4]. Therefore, a checkpoint comprises relevant server state information such as dirty pages, active transactions, and – for the support of instant restart – all acquired locks. The checkpoint is not concerned with the database state and, therefore, it does not write any pages from the buffer pool – as discussed later, this concern is delegated to the page cleaner service.

To take a checkpoint, a BEGIN_CHKPT log record is written to the log. Then, all the required information is gathered from the in-memory data structures and written to the log as so-called checkpoint log records. An END_CHKPT log record is inserted to indicate that the checkpoint completed correctly. The information comprised by a checkpoint summarizes, for the purposes of restoring the server state, all the log records up to the point where the checkpoint began.

After a crash and during system restart, the log analysis phase retrieves the most recent checkpoint and starts a forward scan from there up to the end of the log. During the scan, all log records are analyzed to update the information collected by the last completed checkpoint. As a result, this log analysis delivers the relevant restart information for the state exactly before the crash occurred.

2.3 Page Cleaning

The REDO phase is responsible for replaying all log records that refer to pages marked as dirty in the buffer pool at the moment of system failure. Therefore, the amount of work to be processed by REDO is directly related to two aspects: (1) the amount of dirty pages in the buffer pool at failure time and (2) how old the versions of their persistent page copies are (the older it is, the more log records are expected to be replayed to bring the page to its most recent state).

During normal processing, there are three situations in which pages are flushed from the buffer pool to persistent storage. First, if a dirty page is picked for eviction, it is flushed to persistent storage to making room for fetching the new page. Considering systems with a large buffer pool, the process of evicting a dirty page tends to happen less frequently. Thus, regularly flushing dirty pages at appropriate points is beneficial for reducing the REDO time in case of a crash. This corresponds to the second situation. Third, at normal system shutdown, it is desirable to flush all dirty pages to avoid any recovery at the next start up.

Most modern database systems implement a page cleaner service running as a background thread to handle all these situations. In other words, the task of flushing a page is always delegated to the page cleaner service.

Once it is activated, the page cleaner iterates over the buffer pool and inspects the frame descriptor blocks to determine whether the pages contained should be cleaned, i.e, flushed to persistent storage. Pages can be selected according to a number of policies, e.g., hottest first or oldest first. In this work, we assume a naive policy that simply picks all dirty pages. Pages picked to be cleaned are then copied to a separate buffer; this is done to reduce the time for which a shared latch is held on the page – from a synchronous write call to a fast *memcpy* operation. The pages in the cleaning buffer are then sorted by their page identifier to enable sequential writes. After the write buffer is flushed, the cleaner must once again attempt to acquire shared latches on the pages just flushed and mark them as clean, in case there are no further modifications on such a page made during the cleaning process. In total, each cleaned page is latched three times: to verify its dirty state, to copy it into an in-transit buffer, and to update the dirty state.

2.4 Log Archiving

In WAL-based database systems, the latency of the log device has a direct impact on transaction throughput. Therefore, to enable better performance, latency-optimized stable storage (such as SSDs) for the *recovery log* is usually employed. However, compared to storage devices such as hard-disk drives, latency-optimized devices have a much higher capacity/cost ratio and are therefore less cost-effective. Furthermore, it is even more expensive to keep old log records in a latency-optimized device, assuming that most of these log records are unlikely to be needed. To avoid filling-up the temporary log device, the log records are continuously moved into a *log archive* using a cost- and capacity-optimized, long-term stable storage device. The latency of such a log archive device is not critical for the system performance, since archived log records are usually needed only during recovery from a media failure.

Instead of simply moving the log records from the temporary log to the log archive, it is possible to re-organize them in a more convenient way. To enable single-pass restore of a storage device after a media failure [8], when moving the log records to the archive, they are sorted into runs ordered by page identifier using an external sort-merge operation. As a result, the log archive is said to be *partially sorted*, given that the primary sort criterion is an LSN range – which identifies a run – and the secondary sort criterion is the page identifier. These runs are merged asynchronously and incrementally, using a data structure similar to a partitioned B-tree [1].

The partial sort order in the log archive also enables indexing, meaning that the history of a single page can be accessed much more efficiently. This was exploited in a proposal for instant restore [9], which enables on-demand restoration from a media failure concurrently with running transactions, similar to instant restart. However, such indexing can also be exploited for efficient REDO operations during restart after a system failure as well as for single-page recovery in case of an isolated data corruption [3]. In our case, we exploit the partially sorted log archive for a novel application: propagation of updates with a decoupled page cleaner, which is discussed in the next section.

3. DECOUPLED PERSISTENCE SERVICES

3.1 Decoupled Checkpoints

The idea behind a decoupled checkpoint is to use the same logic behind log analysis to gather all information needed for the checkpoint, instead of querying in-memory data structures for this reason. To take a decoupled checkpoint, all log records between the last completed checkpoint and the BEGIN_CHKPT log record referring to the checkpoint currently being taken are analyzed. The main motivation here is that, as minimal as it may be, any interference with the data structures caused by the process of checkpoint creation can be completely avoided. However, since it is desirable to re-use the same logic of log analysis, some important differences must be considered.

First, taking a traditional checkpoint is completely independent from information contained in older checkpoints, since it only inspects in-memory data structures. However, in the case of a decoupled checkpoint, since the algorithm is the same as for log analysis, it must rely on information present in the checkpoint previously completed. This introduces the limitation that, when taking a new decoupled checkpoint, the previously completed checkpoint must always be contained in the log. Alternatively, the contents of such a checkpoint can be cached in main memory during normal processing.

Second, since the decoupled checkpoint inspects only log records, page write operations must be logged to determine when a previously dirty page can be considered clean. As observed in previous studies [7], logging page writes is a common practice used in existing database software, since it enables more effective recovery from a system failure.

Third, taking a decoupled checkpoint may introduce additional I/O for reading the temporary log. An approach that eliminates this overhead maintains checkpoint information in main memory and continuously updates it by consuming log records from the log buffer, which also resides in main memory. When a checkpoint is requested, this information can then immediately be propagated to the persistent log. Additionally, the checkpoint generation process can feed from the same input stream used for log archiving. When these two techniques are combined, no additional I/O overhead is incurred. For simplicity of implementation, our evaluation considers a naive approach where checkpoints always rely on reading the persistent log.

Fourth, since the process of taking a decoupled checkpoint is basically the same as log analysis, the longer the elapsed time from the last checkpoint, the more time is required for taking a new decoupled checkpoint. Therefore, the new technique encourages more frequent checkpoints, which also benefits recovery performance. Since this kind of checkpoint creation does not interfere with in-memory processing, overall system performance should not be affected.

3.2 Decoupled Page Cleaner

To motivate the use of a decoupled page cleaner, we are first considering a scenario where the whole working set is in main memory, later expanding our argument to medium and small buffer pools. For such a large buffer pool, there are no page misses and therefore no need for page eviction. Therefore, the only I/O operations to the database device made by the cleaner service are related to periodically cleaning dirty pages, which reduces REDO time during restart after a system failure. Hence, it is desirable for the cleaner service to be continuously active and provide for as many clean pages as possible in the buffer pool.

To flush pages, the cleaner service requires direct access to the buffer-pool data structure as well as unnecessarily high latching (i.e., low-level concurrency control) contention. If the page cleaner service is too aggressive, it might generate too much interference (mainly with hot pages) and consequently harm the transaction activity. Hence, it is highly desirable to reduce any interference with in-memory data structures while enabling the page cleaner to run as aggressively as required. Note that this stands at odds with the goal stated above, which means that a tradeoff is required in the traditional design.

A *decoupled* page cleaner eliminates this trade-off by encouraging more frequent checkpoints regardless of transaction activity. This is achieved by asynchronously replaying log records on page versions present on persistent storage, without requiring any access to the page images in the buffer pool. This is a similar procedure as those performed by single-pass restore and virtual backups [8].

However, assuming a group of sequential pages that are in the cleaner buffer, fetching log records referring to these pages implies random I/O to the recovery log device, which should be avoided as far as possible for performance reasons. Fortunately, as mentioned in Section 2.4, the log archive device can be partially sorted by page identifier and indexed, enabling a sequential read of the log records referring to the pages to be cleaned. Furthermore, to avoid repeated work, the decoupled page cleaner keeps track of the LSN up to which pages were cleaned. Every activation of the decoupled cleaner then increases this LSN value, essentially bringing the persistent database state up to that LSN. Figure 2 illustrates the main idea of the decoupled page cleaner based on log archive.

Two further issues must also be considered to fully decouple a page cleaner from the buffer pool. First, an efficient eviction policy should give preference for removing clean pages from the buffer to make room for new pages. Therefore, to keep track of which pages are dirty and which pages are clean, the decoupled page cleaner, after replaying the log records and flushing a more recent version of pages, still has to acquire latches on pages in the buffer pool to mark them as clean, if necessary. Second, when a page flush is required and the decoupled cleaner is activated, it must be guaranteed that the page is in its most recent version when fetching it from persistent storage again. This requires some synchronization among the page cleaner and the process of fetching pages from disk.

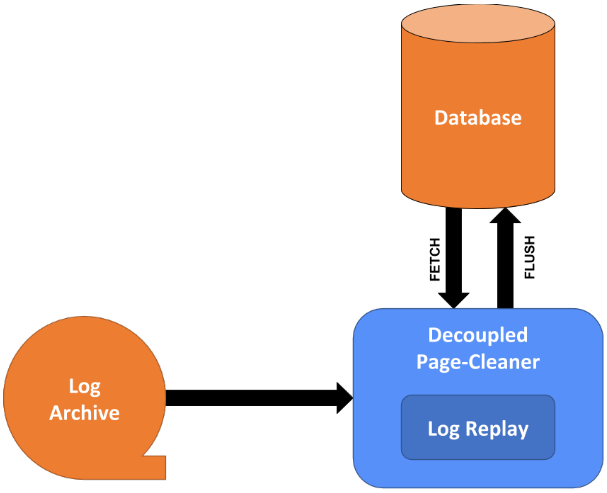


Figure 2: Decoupled cleaning scenario

To overcome these limitations, the decoupled page cleaner can be combined with a method used for single-page recovery [3]. By employing this kind of recovery, it is possible to determine whether or not a page fetched from persistent storage is up to date. In case the page is outdated, log replay is triggered to bring it to the most recent state before allowing further accesses. This addresses the second problem state above. A similar idea, called *write elision*, can be employed to deal with the eviction problem. It relies on single-page recovery to allow evicting a dirty page from the buffer pool without writing it back to disk. In combination, these techniques enable full decoupling of persistence services from buffer-pool data structures.

In a scenario where the working set does not fit into main memory, the advantages of a decoupled cleaner must be reconsidered. If a medium-sized buffer pool is used, such that memory bandwidth is well utilized for high transaction throughput, the decoupled cleaner is not expected to impact performance in any way – positive or negative. However, the advantage of a modular and loosely-coupled architecture suggests it as a better approach. In the case of a small buffer pool, where page I/O clearly becomes the bottleneck, a traditional approach is preferrable, since it can more efficiently provide propagation solely by page eviciton – i.e., without an asynchronous cleaning daemon. This scenario, however, is quite rare in modern OLTP systems, given the moderate cost of DRAM.

3.2.1 Further Applications of Decoupled Cleaning

Other than decoupling the modules of page cleaning and buffer management in the system architecture, a decoupled page cleaner enables interesting recovery features which we hope to exploit in future work. These are not considered in our evaluation in Section 4, but are shortly listed here to motivate further applications of our technique.

Similarly to write elision, a decoupled cleaner together with single-page recovery also enables *read elision* [2]. The idea consists of simply logging insertions and updates to database pages without the requirement of having them loaded from persistent storage to memory. The decoupled page cleaner is then responsible for later replaying the generated log records to the persistent pages. Both read and write elision offer the advantage of avoiding unnecessary I/O operations that would increase the transaction response time. In other words, the decoupled page cleaner works in synergy with single-page recovery in the sense that it runs page recovery in the background and alleviates the cost of doing it on demand.

A decoupled cleaner also allows different page representations for buffered and persistent images of pages, which may exhibit several advantages. For instance, in a design that maintains only committed log records in the persistent log [10], a *no-steal* policy is easily achieved. In such a design, uncommitted updates always remain in the volatile buffer pool, so that the need for UNDO logging & recovery is eliminated.

4. EXPERIMENTS

As the main hypothesis to be tested by the following experiments, a decoupled design may reduce the interference with main-memory data structures, which might deteriorate the transaction throughput of the whole system. In our experiments, we only evaluate the performance of decoupled checkpoints. The decoupled cleaner provides similar results, but a more thorough analysis is currently being carried out and it is thus reserved for a future publication.

We implemented the algorithms previously described for creating decoupled checkpoints in the Zero storage manager[1] (based on Shore-MT [5]). Here, we present experiments comparing the classical and the decoupled checkpoint implementation to exemplify how such a service can interfere with in-memory processing. The experiments consist of executing the TPC-B benchmark, which performs a large amount of small read-write transactions, such as debit and credit on a bank account. For each experiment, the database is loaded and the benchmark is executed for 15 minutes.

The scale factor of TPC-B is set to meet the exact number of hardware threads in such a way that each hardware thread executes transactions referring to a single branch. Consequently, there are no concurrency conflicts that might interfere with the transaction throughput. Furthermore, in order to simulate modern in-memory database systems, the buffer pool size is set to fit the whole database size after executing the benchmark. Database and log archive are each stored in its own latency-optimized device (SSD). The experiments were executed with the recovery log both in a dedicated latency-optimized device (SSD) and in a main-memory file system. Having the recovery log in a main-memory file system enables a simulation of database systems with higher transaction throughput, where interferences should have a larger impact on performance. However, this approach does not represent a real-world scenario, since a recovery log must always be stored on a non-volatile device.

The experiments with decoupled checkpoints were executed with log archiving disabled. Transaction throughput was measured by inspecting the log records generated during the benchmark execution.

Finally, the experiments described here were carried out on an Intel Xeon X5670 server with 96 GB of 1333 MHz DDR3 memory. The system provides dual 6-core CPUs with hyper-threading capabilities, which gives a total of 24 hardware thread contexts. The operating system is a 64-bit Ubuntu Linux 12.04 with Kernel version 3.11.0.

Figure 3a shows the results after running the benchmark with the log being on an SSD device. Values on the y-axis represent the average transaction throughput in units of thousand transactions per second. Values on the x-axis represent how frequent a checkpoint request is made in milliseconds. Taking checkpoints too frequently increases the interference with the in-memory data structures, which may harm the transaction throughput of the system. Even though taking a checkpoint every millisecond is not a realistic scenario, it is done in order to stress the system and enforce as much interference as possible.

The transaction throughput of decoupled checkpoints is not only higher when compared to classical checkpoints, but the variation of throughput between different checkpoint frequencies is smaller. Ideally, the throughput of decoupled checkpoints should be constant for any checkpoint frequency, since there is no interference with in-memory data structures that might disturb the system performance. However, the additional I/O operations required for a decoupled checkpoint to read log information might induce a delay on the writing of log records for transaction commit.

Taking classical checkpoints every interval higher than 1000 ms does not produce significant interference in the system and throughput after this point equals the one achieved by decoupled checkpoints.

The CPU consumption for the whole benchmark execution was also analyzed. The higher the transaction throughput, the higher is the CPU consumption. In Figure 4, we compare both designs in the case previously mentioned where the transaction throughput is the same. Here we can observe that decoupled checkpoints consume a small extra amount of CPU compared to classical checkpoints.

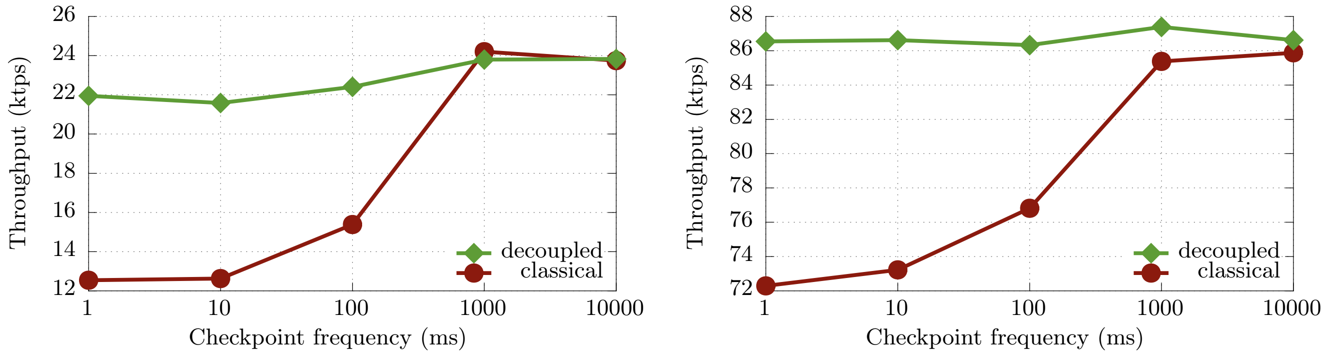
5. SUMMARY AND CONCLUSIONS

In this contribution, we propose an alternative architecture for database systems by decoupling the propagation components from in-memory processing. Decoupling components is desirable not only because it avoids any interference with in-memory data structures, but also because it eliminates much of the code complexity introduced by unnecessary interactions between components. To that end, we implemented novel checkpoint and page cleaner algorithms that are based on log information rather than on data collected from critical in-memory data structures. Based on a qualitative analysis, we verified that the complexity of our codebase was significantly reduced.

Our techniques are better suited for scenarios where the majority of the application working set fits into main memory. This is because decoupled persistence services are assumed to have a more proactive and preventive role – rather than being critical components on which transactions depend in order to make progress. However, the techniques are not restricted to in-memory workloads in any way.

The experiments presented in this work show that the

[1] https://github.com/caetanosauer/zero



(a) Log in SSD

(b) Log in RAM

Figure 3: TPC-B throughput.

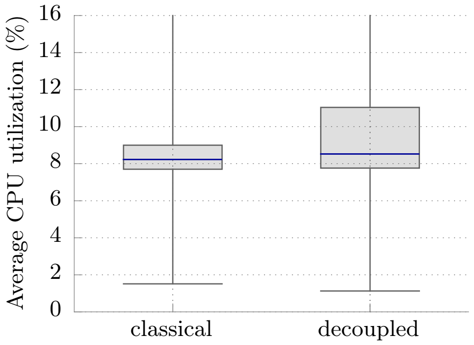


Figure 4: CPU consumption of Figure 3a with 10 sec.

decoupled strategy for checkpoints improves the transaction throughput of the system when persistence services are executed very frequently. As the frequency is reduced, the performance of traditional techniques is matched, but the decoupled strategy does not incur any performance penalty. For future work, we plan to evaluate our decoupled page cleaner in more detail, including an analysis of different cleaning policies for both classical and decoupled services. We believe that a combination of the techniques can achieve the most effectiveness in reducing recovery time, which is the main goal of page cleaning.

6. REFERENCES

[1] G. Graefe. Sorting and indexing with partitioned b-trees. In *Proc. CIDR*, pages 5–8, 2003.

[2] G. Graefe, W. Guy, and C. Sauer. *Instant recovery with write-ahead logging: page repair, system restart, and media restore*. Synthesis Lectures on Data Management. Morgan & Claypool Publ., 2014.

[3] G. Graefe and H. A. Kuno. Definition, detection, and recovery of single-page failures, a fourth class of database failures. *PVLDB*, 5(7):646–655, 2012.

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

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

[6] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.

[7] C. Sauer, G. Graefe, and T. Härder. An empirical analysis of database recovery costs. In *Proc. RDSS Workshop (co-located with SIGMOD)*, 2014.

[8] C. Sauer, G. Graefe, and T. Härder. Single-pass restore after a media failure. In *Proc. BTW, LNI 241*, pages 217–236, 2015.

[9] C. Sauer, G. Graefe, and T. Härder. Instant restore after a media failure, 2016. *Submitted for publication.*

[10] C. Sauer and T. Härder. A novel recovery mechanism enabling fine-granularity locking and fast, redo-only recovery. *CoRR*, abs/1409.3682, 2014.

[11] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. VLDB*, pages 1150–1160, 2007.