TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Decoupled Propagation for DBMS Architectures

by

Lucas Lersch

A thesis submitted in partial fulfillment for the degree of
Master of Science

in the
Fachbereich Informatik
AG Datenbanken und Informationssysteme

December 2015

Declaration of Authorship

I, Lucas Lersch, declare that this thesis titled, 'Decoupled Propagation for DBMS Architectures' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ____________________________________

Date: ____________________________________

"To raise new questions, new possibilities, to regard old problems from a new angle, requires creative imagination and marks real advance in science."

- Albert Einstein

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Abstract

Fachbereich Informatik
AG Datenbanken und Informationssysteme

Master of Science

by Lucas Lersch

This thesis implements and evaluates an alternative decoupled design for propagation services in a database system architecture. The substantial fall in the cost of volatile memory in the past years created the need for system architectures that can take full benefit of large buffer pools. Even with large amounts of memory, it is still desirable for classical database architectures to keep an up-to-date version of database pages on stable storage, in order to support efficient recovery. Therefore, we consider the existing components of a classical database architecture and chances for optimization in the context of persistence and system recovery. Decoupling propagation components from in-memory processing enables a simpler and more modular architecture, as well as less interference from persistence services in critical in-memory data structures. The decoupled design proposed in this work includes novel checkpointing and page cleaning algorithms that are based on log information rather than on data collected from critical in-memory data structures, such as buffer pool and transaction tables.

Acknowledgements

I would like to express my gratitude to my advisor Caetano Sauer. Not only his guidance was essential for this work, but his patience and openness made the whole process enjoyable. It is a honor and a pleasure to have worked with Caetano for the past years and witnessed a bit of his brilliance.

I would like to thank Prof. Dr. Theo Härder for the opportunity to work in his research group and for all the assistance given to me and so many other students. It is admirable and inspiring how a single person can have such a positive impact in the lives of so many people.

I would also like to thank Prof. Dr. Stefan Deßloch for his openness and kindness while being the mentor of my master studies in Technical University of Kaiserslautern. I am grateful for all the colleagues and friends that shared the same work place for the past years. My special thanks goes to Pedro, Vitor and Gilson.

Finally, I would like to offer my thanks to all my family and friends. I am particularly grateful to my parents, Nelson and Suzana, for the constant support they have given me during all my life.

Contents

Dedicated to Nelson and Suzana

Chapter 1

Introduction

1.1 Motivation

The architecture of classical database systems was conceived in a time when the available amount of main memory was relatively small compared to the size of typical application working sets. Since these systems must be heavily optimized for the underlying hardware, the design had to consider frequent operations to persistent storage, taking into account the high latency of both commit operations and data accesses. However, advances in hardware technology usually require an adaptation of the existing software architecture in order to deliver optimal performance [1].

One of the most important hardware improvements is the dramatically fall in the cost of volatile memory in comparison to 30 years ago, as seen in Figure 1.1. Therefore, nowadays, it is realistic to consider a scenario where most, if not all, data pages fit in the buffer pool and, consequently, there are fewer page reads and writes 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 classical hard drive disks. In this work, we reconsider the existing components of classical database architecture and chances for optimization in the context of persistence and system recovery.

We assume that even with large amounts of memory and most of transaction processing not requiring I/O operations, it is still desirable to keep an up-to-date version of database pages in persistent storage, in order to guarantee efficient system recovery in the case of a system failure. Most modern database systems rely on the write-ahead logging technique and implement the ARIES [2] algorithm for system recovery. In order to improve recovery performance, regular checkpoints are taken. Checkpoints serve as the starting point for log analysis, the first phase of crash recovery. The more recently a checkpoint was taken, the less time the log analysis phase takes to complete in case of a failure. In addition to checkpoints, it is also common for

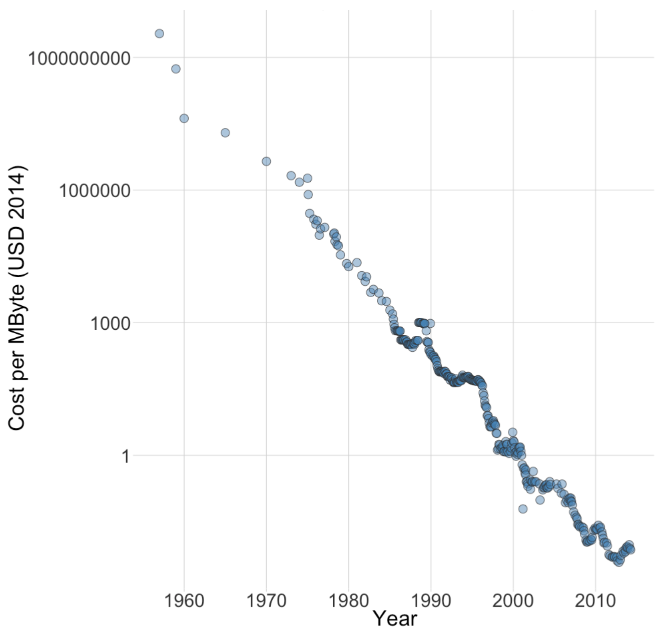
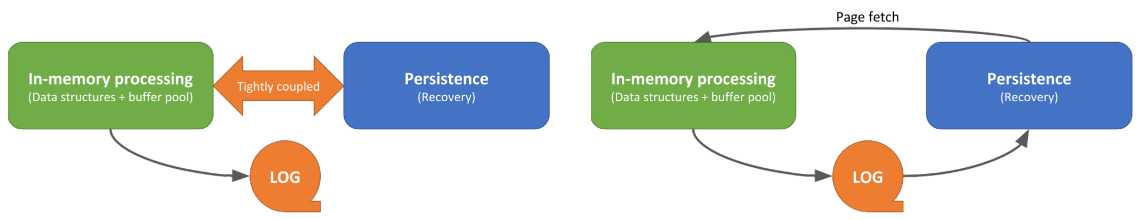


Figure 1.1: The falling cost of memory [4]

database systems to employ another background thread to periodically clean dirty pages by flushing them from the in-memory buffer pool to persistent storage. Consequently, since the REDO phase would require random reads, its cost is reduced by maintaining a low count of dirty pages [3]. In this work, we refer to both the checkpoint and the 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 and might disturb in-memory transaction processing, since they must inspect data structures and consequently acquire and release latches. Even if this interference might not be crucial to the performance of the whole system, it can be avoided. This behaviour is highly desirable, since in modern systems main-memory is abundant and page I/O is not the bottleneck any more, in-memory performance and scalability are crucial. For this reason, the “interference” we talk about becomes much more relevant. Furthermore, decoupling unnecessary coupled components in a system is not only a matter of performance, but it also offers benefits from the architectural design and code complexity point of view, since the in-memory structure implementation does not have to worry about a checkpoint possibly querying its state at an arbitrary point, for example. Therefore, the major goal of a decoupled design is to provide a higher degree of modularity and enable the reuse of the architectural components of a system.



(A) Tightly coupled persistence (B) Decoupled persistence

FIGURE 1.2: Differences between classical and proposed architecture.

1.2 Contribution

The general contribution of this work is to propose an alternative architecture for propagation services in a database system. The main idea is to decouple all propagation aspects from in-memory database data structures (buffer pool, lock manager, transaction table) an rely solely on information from the log. Figure 1.2 illustrates differences between the classical design and the proposed decoupled design.

The first contribution of the proposed architecture is to enable checkpoints to gather all required information solely by inspecting the log. The advantage is that it is not only made simpler by making use of the same existing logic behind log analysis, but it also implies less interference in in-memory processing, as mentioned above.

The second contribution is a decoupled implementation of a page cleaner. Besides also reducing the interference in the buffer pool, it can be combined with single-page recovery techniques [5] that enable a set of interesting features for the recovery component of a database system, as discussed later in this thesis.

The remaining of this work is organized as follows. Chapter 2 provides the background theory required for the further understanding of this work. The concept of checkpoints is discussed as well as relevant details about page cleaning, logging, log archiving, and ARIES system recovery. Chapter 3 describes the requirements and implementations of a decoupled design for checkpoints. Differences, advantages, and problems faced are as well discussed. Chapter 4 describes in detail a page cleaning service based on log replay. Limitations and important aspects are also discussed. Chapter 5 defines the environment setup, and methodologies used for running the experiments. Also, the found results and findings are presented and discussed. as well as the results and findings. Finally, Chapter 6 concludes this work by summarizing our contribution and providing guidelines for future work.

Chapter 2

Background Theory

2.1 System Recovery

In order to offer transaction atomicity and durability guarantees, i.e., the "A" and "D" of ACID, most modern database systems implement the write-ahead logging mechanism. The ARIES algorithm [2] defines an efficient way to enable system recovery using write-ahead logging. In the case of a failure, the goal of system recovery is to re-establish the system as it was immediately before the failure happened.

Here we differentiate the concepts of *system state* and *database state*. The system state comprises information that does not refer directly to data contained in database pages. The states of in-memory data structures, e.g., which pages are present in the buffer pool, or which transactions are active in the transaction table, are part of the system state. The database state is then defined as all information that is contained in database pages. When a system failure occurs, all information on volatile memory is lost at restart, and the system recovery process kicks in. At the end of system recovery, both system state and database state are at the most recent consistent state.

The system recovery algorithm operates in three different phases: log analysis, REDO, and UNDO. Log analysis employs a forward-scan in the log to inspect log records in order to determine which pages were dirty and which transactions were active. Alternatively, a backward-scan could be employed to simplify the log-analysis algorithm. In this case, if a transaction commit log record is found, it is guaranteed that it is not a loser transaction and it is safe to ignore all the following log records referring to the same transaction. Independently of the scan direction, the information generated by the log analysis phase is the same and is then used by both REDO and UNDO phases.

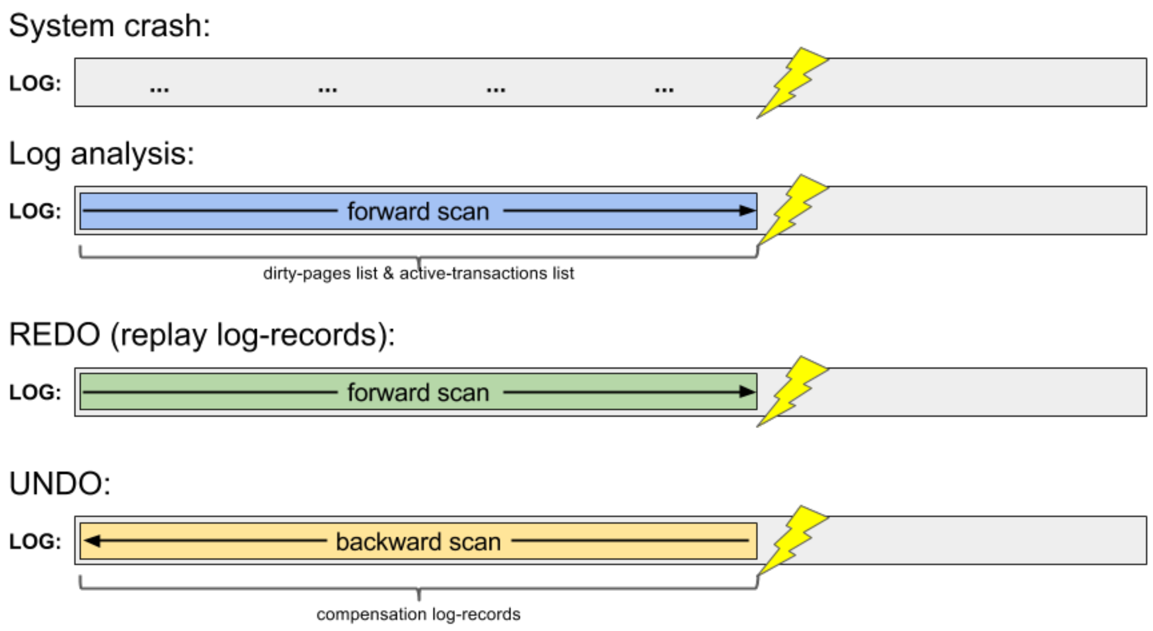


FIGURE 2.1: Phases of the recovery algorithm.

After log analysis, the REDO phase gets a list of which pages were dirty at the time of failure. Each entry in this list contains, besides the page identifier, also the DirtyLSN, i.e., the LSN of the log record of the oldest non-propagated update to that page. The oldest (minimum) DirtyLSN of all pages in the list defines the point where REDO has to begin a forward log-scan until the most recent log record. From there, all the log records that refer to updates to dirty pages are replayed.

Similarly, the UNDO phase gets a list of transactions active at the time of failure. As opposed to log analysis and REDO, UNDO employs a backward log-scan starting from the most recent log record. Each transaction in the list contains the FirstLSN, which refers to the LSN of the first log record of that transaction. The stop point of UNDO is determined based on the oldest FirstLSN among all active transactions. During the backward log-scan, each log record referring to a change made by an active transaction is undone and a compensation log record is generated. Figure **??** summarizes the behaviour of each phase after a system failure occurs.

It is important no note that during recovery the system is halted and not accepting transactions. *Instant restart* [6] presents an alternative to the classical recovery algorithm mentioned above, enabling new transactions to be accepted as soon as the log analysis phase complete.

2.1.1 Instant Restart

The main idea to open the system for new transactions while still doing recovery is to enable on-demand REDO and UNDO. In a first scenario, to open the system after REDO, we have

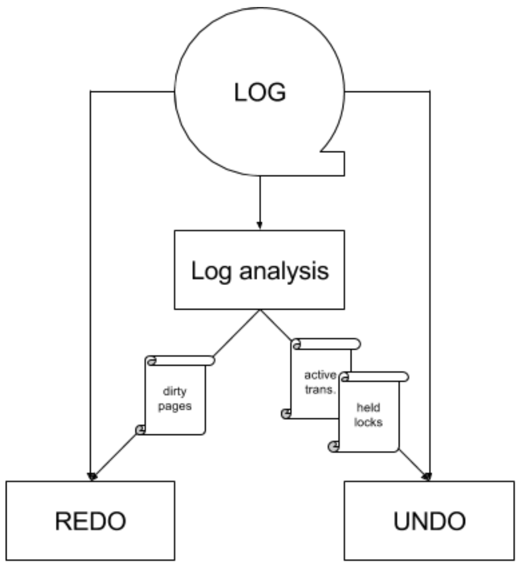


FIGURE 2.2: The recovery algorithm

to enable UNDO to be performed by means of concurrent transaction roll-back instead of a log-based scan. In this case, the loser transactions to be rolled-back must reacquire their locks prior to UNDO in order to protect data from the new transactions. In the context of ARIES, such locks are collected during the REDO phase [2]. If a new transaction attempts to acquire a lock being held by one of the loser transactions, this would trigger the roll-back of the loser transaction by undoing its chain of log records.

In a more complete scenario, the system is opened for new transactions after log analysis. In order to do that, REDO must be able to be performed by means of on-demand single-page recovery. Similar to on-demand UNDO, whenever a new transaction tries to access a page that was dirty but is not yet recovered, the REDO of that single page is done by following the chain of log records referring to that same page. In this case, the locks required for on-demand UNDO must be reacquired during log analysis instead of REDO, as mentioned above. Figure 2.2 illustrates the relation between the log and the phases of the recovery algorithm.

Finally, it is possible to note that the performance of system recovery is directly related to the log size, since it must be inspected many times. A system that has been running for hours and generated a large amount of log records could take at least some few hours to be recovered if the log was to be read from the very beginning. It is highly desirable to have mechanisms to speed up the phases of the system recovery process. This can be achieved by taking regular checkpoints, as discussed below.

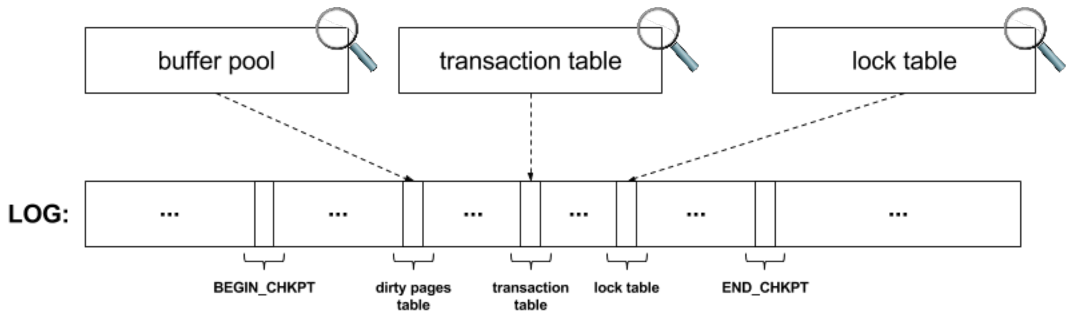


FIGURE 2.3: Taking a checkpoint

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 with relation to the system failure point in time, the less work is required for recovering the system. Thus, it is a good practice to take checkpoints regularly. In order to be more precise in the definition of a checkpoint, this work is going to follow the definition of *fuzzy checkpoints* as presented in [7]. Therefore, a checkpoint comprises relevant information that defines the system state, such as:

- list of which pages in the buffer pool are marked as dirty
- list of which transactions in the transactions table are currently active
- list of which locks in the lock table are currently being held

To take a checkpoint, a BEGIN_CHECKPOINT 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 checkpoint log records. An END_CHECKPOINT log record is inserted to indicate that the checkpoint completed correctly. The information comprised by a checkpoint summarizes all the log records up to the point where the checkpoint was taken. Figure 2.3 illustrates how checkpoint log records are organized among other log records.

After writing the END_CHECKPOINT log record, the LSN of the BEGIN_CHECKPOINT log record is written to a separated storage location. This is called the MasterLSN. Writing the END_CHECKPOINT log record and updating the MasterLSN in the file do not have to be made as single atomic-operation. In case of system failure in between, the only consequence is that the log-analysis phase will start from an older checkpoint, and the completed and most recent one is going to be ignored. Saving the MasterLSN is not required if log analysis employs a backward-scan (as mentioned above), since it is always possible to find the most recent checkpoint.

It is important to note that the checkpoint information is being gathered while transactions are still running, i.e., the system is not halted for this purpose. As a consequence, a fuzzy checkpoint does not necessarily represent any system state that existed at some point in time. For example, a checkpoint might indicate that transactions A and B are active, but while the checkpoint was being taken the transaction A committed and transaction B started, meaning there was no point in time that transaction A and B were both active at the same time.

After a failure and during system restart, the log-analysis phase retrieves the MasterLSN from the determined storage location and starts a forward scan from there up to the end of the log. During the scan, the log records are analyzed in order to update the system state defined by the last completed checkpoint and bring it to the system state exactly before the failure occurred. Therefore, it is not a problem for the checkpoint itself not being consistent, since all eventual inconsistencies are updated when analyzing the following log records. It is also worth mentioning that, in the best case scenario, a checkpoint was successfully taken exactly before the system failure and the only work the log-analysis phase will have is to scan the log for reading the checkpoint log records.

Finally, by our definition, a checkpoint does not directly flush any dirty pages from the buffer pool to persistent storage. The only concern of a checkpoint is to reduce the time spent doing log analysis, having no direct impact in the performance of other recovery phases. In order to improve the performance of REDO, it is required to maintain a low count of dirty pages in the buffer pool. This is addressed by the page cleaner service, as discussed in the next section.

2.3 Page Cleaning

The REDO phase is responsible for replaying 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 the moment of failure and (2) how old the version of the persistent copy of a page is (the older it is, the more log records must be replayed to bring the page to its most recent state).

There are three situations in which pages are flushed from the buffer pool into persistent storage. First, if a dirty page is picked for eviction, it is flushed to persistent storage in order to free space for fetching the new page. Second, considering systems with 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 REDO time in case of a system failure. Third, at normal system shutdown, it is desirable to flush all dirty pages to avoid any recovery at the next start up.

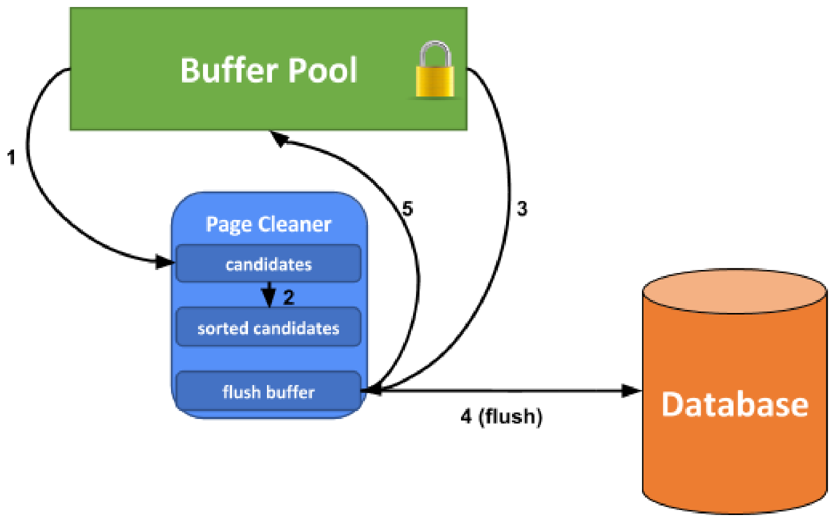


FIGURE 2.4: Page cleaner execution steps.

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. This page cleaner can be either pro-actively activated or receive asynchronous activation signals at points the database system considers cleaning would be appropriate considering event-based policies [8]. In addition to "when" pages should be cleaned, the page cleaner should also determine which pages to clean. Different policies can be applied here as well, such as flushing older pages or hot pages first. Analyzing such policies, however, is not in the scope of this thesis [8].

Once it is activated, the page cleaner iterates over the buffer pool, acquiring shared latches at each page in order to analyze its contents and determine if the page should be cleaned, i.e, flushed to persistent storage. Pages picked to be cleaned are then copied to an internal "candidates" buffer and the shared latch is released. The pages in the cleaning buffer are then sorted by their page identifier, in order to enable sequential writes. Once again, the cleaner tries to acquire a shared latch for the pages in the buffer pool to be cleaned and, if succeeded, copies the most recent version of the page to a flush buffer to be written to persistent storage. 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 changes. Figure 2.4 illustrates these steps in the order they happen.

Finally, it is important to note that the main purpose of the page cleaner is to reduce the amount of dirty pages in the buffer pool, consequently reducing the amount of work required by the REDO phase in the case of a system failure.

2.4 Logging and Archiving

In database systems implementing write-ahead logging, the latency of the log device has direct impact on the transaction throughput. Therefore, in order to enable better performance, latency-optimized stable storage (such as SSDs) for the *recovery log* is usually employed. The problem is that latency-optimized devices have a much higher capacity/cost ratio in comparison to storage devices such as hard drives. Furthermore, it becomes expensive to keep old log records in a latency-optimized device, even more if we consider that most of these log records are unlikely to be needed. In order to avoid the temporary log device to fill up, log records are moved to a *log archive* in a cost- and capacity-optimized, long-term stable storage device. The latency of the log archive device is not critical for the system performance, since archived log records tend to be needed only in the context of restoring the main storage device after a media failure.

Instead of simply moving the log records from the temporary log to the log archive, it is possible to organize them in a more convenient way. In order to enable a single-pass restore of a storage device after a media failure [9], when moving the log records to the archive, they are sorted by page identifier using an external merge sort operation. As a result, the log archive is said to be partially sorted. In other words, the log archive comprises a set of runs, where each run contains log records in a certain log sequence number interval and the log records within a run are primarily sorted by its page identifier.

Besides that, only log records with page identifier, i.e., those representing modifications, are selected for archiving. In other words, only page-update log records are archived. This is aligned with the assumption that the log archive only supports REDO, whereas UNDO relies on the recovery log.

Finally, even in the log archive, accessing random log records to perform log replay might result in too many I/O operations. However, due to the partially-sorted organization of the log archive, it is possible to have much faster indexed access to the desired log records. Figure 2.5 illustrates the idea behind the described log archive. With efficient access to log records and support for REDO, the log archive has the potential to be used in the REDO phase of restart, rather than only for restore. This assumption becomes the main idea for enabling a decoupled page cleaner service, as later described in this work.

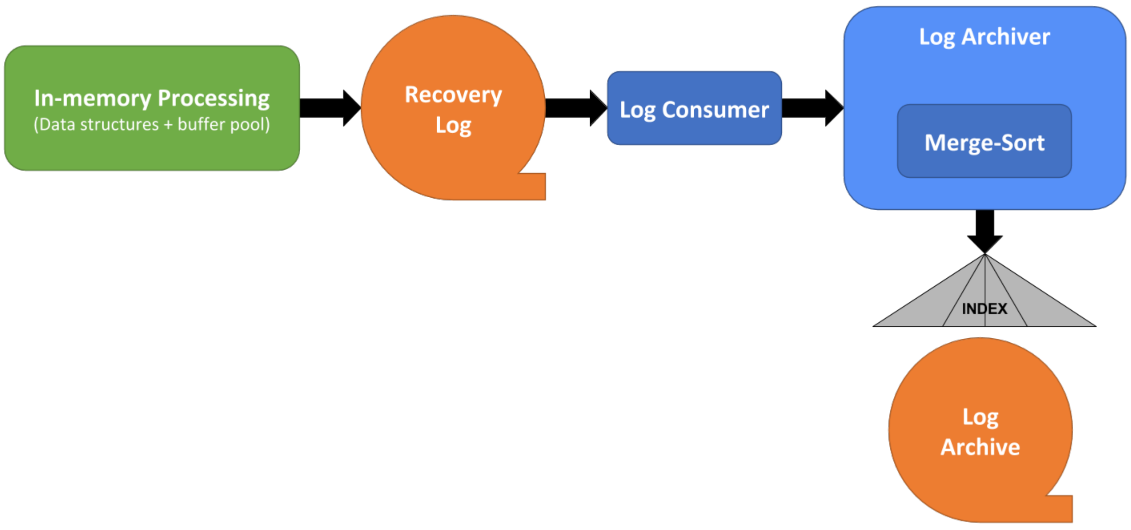


FIGURE 2.5: Partially-sorted Log Archive.

Chapter 3

Decoupled Checkpoints

3.1 Main Idea

As mentioned in Section 2.2, the main purpose of the log analysis phase is to simply update the system state contained in the most recent checkpoint by analysing the log records after the BEGIN_CHECKPOINT. When a request for taking a checkpoint is made, an in-memory object that represents the system state is instantiated. The basic structure of this object is as follows:

```
struct chkpt_t{
    uint64_t begin_lsn;

    uint16_t next_volume_id;
    map<...> mounted_devices_table;

    map<...> dirty_pages_table;

    map<...> acquired_locks_table;

    uint64_t youngest_transaction_id;
    map<...> active_transactions_table;
};
```

The idea behind a decoupled checkpoint is to use the same logic behind log analysis to populate the checkpoint object itself, instead of querying in-memory data structure for the required information. In other words, the log analysis phase of system recovery is modified to generate a checkpoint object representing the most recent system state. Figure 3.1 illustrates the basic difference between the classical checkpoint and the decoupled checkpoint. The main motivation here is that, as minimal as it may be, any interference in the data structures caused by the process of taking a checkpoint can be completely avoided. However, since it is desirable to re-use the same logic of log analysis, some important differences must be considered. This differences are discussed in the next section.

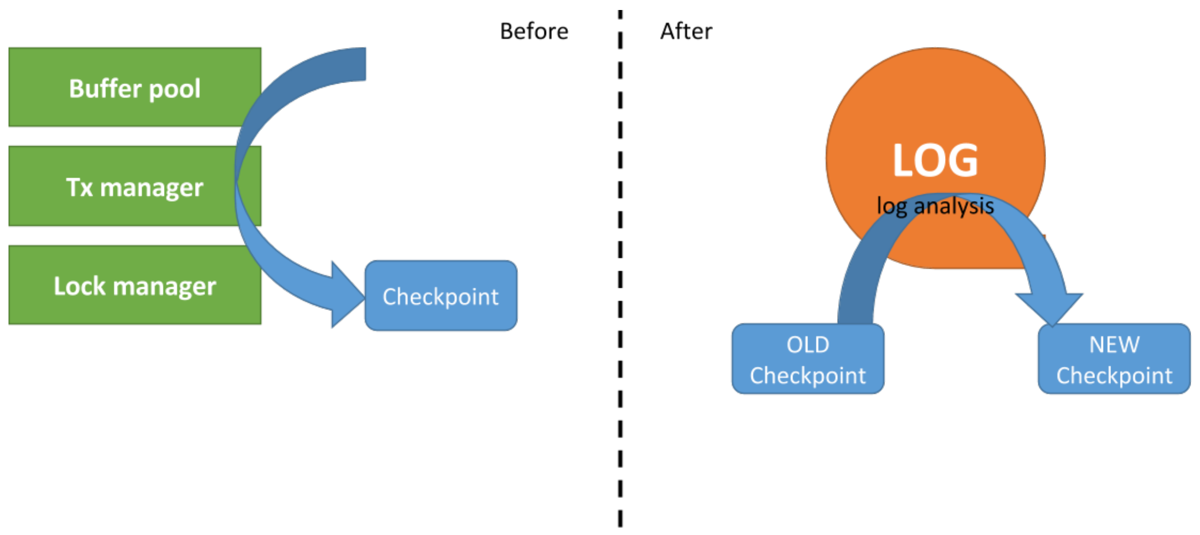


FIGURE 3.1: Differences between classical checkpoint and decoupled checkpoint.

3.2 Implementation Details

First, every time a classical checkpoint is taken, is it completely independent from information contained in older checkpoints, since it only relies on information extracted from the in-memory data structures. However, when a decoupled checkpoint is being taken, since the algorithm is the same as log analysis, it relies on information contained in the previous checkpoint to capture the current system-state. This introduces the limitation that, when taking a new decoupled checkpoint, the previous completed checkpoint must always be in the log. Even if checkpoints are taken regularly and it is unlikely that reclaiming log space would erase the previous completed checkpoint, our design assumes that the log space reclamation policy guarantees that such situation does not happen.

Second, when log analysis is scanning the log for recovery, the system is halted and no log records are being generated. Thus, it is possible to scan the log until the end, even though this is not true when taking a checkpoint. Therefore, an upper-limit for the forward-scan must be defined to indicate where the log analysis should stop. This is simply solved by using the BEGIN_CHECKPOINT of the checkpoint currently being taken as the upper limit. It is possible to conclude that, as in a classical checkpoint, a decoupled checkpoint will summarize all log records that happened before its BEGIN_CHECKPOINT log record. Figure 3.2 illustrates this idea.

Third, since a classical checkpoint inspects the buffer pool, it is able to implicitly determine which pages were cleaned between one older checkpoint and the current one. Decoupled checkpoints have no way to determine if a dirty page was cleaned. Thus, in order to enable a decoupled checkpoint, it is necessary to log page flush operations to indicate which pages were cleaned. Furthermore, the decoupled checkpoint implementation is orthogonal to the direction of the

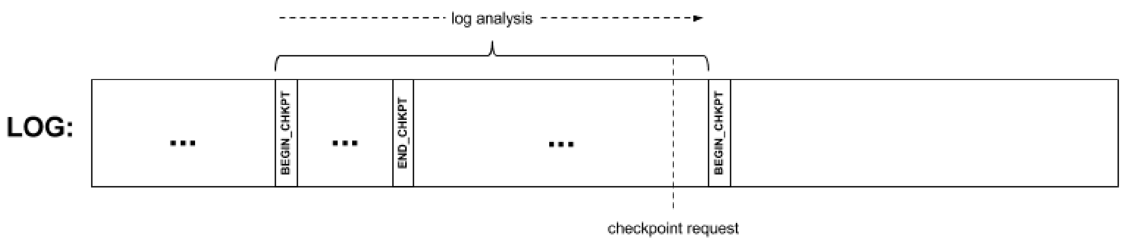


Figure 3.2: Log-based decoupled checkpoint.

scan employed by log analysis. However, backward log analysis could benefit from these page-write log records in a similar way it does from transaction-commit log records, as mentioned in Section 2.1. If a page-write log record is found, it is guaranteed that the page was flushed at that moment and it is safe to ignore any older log record referring to updates to that same page. It is assumed, for simplicity, that log analysis employs the classical forward scan.

Also, it is important to mention that when taking a decoupled checkpoint, only the information from the temporary log is read, not the log archive. Reading this information is done in a buffered way, in order to avoid reading one log record at a time and provoke too many I/O operations. Classical checkpoints, on the other hand, do not require any I/O. In terms of decoupled architecture, it is possible to abstract the interactions between components of a database system to a producer-consumer pattern. In this abstraction, it is considered that in-memory transaction processing produces log records which can then be consumed by services such as the log archiver and decoupled checkpoint. Figure 3.3 illustrates this abstraction.

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. Furthermore, even if more CPU time is required for analysing the log, our assumption is that in most modern database systems CPU tends to be idle most of the time, due to aspects such as lock contention. Finally, modern database systems tend to have very large memory and run thousands of transactions per second, implying that the in-memory data structures (buffer pool, transactions table, locks table) tend to be larger. The possible slowest performance and additional CPU time are prices worth paying for generating no interference with the processing in these in-memory data structures.

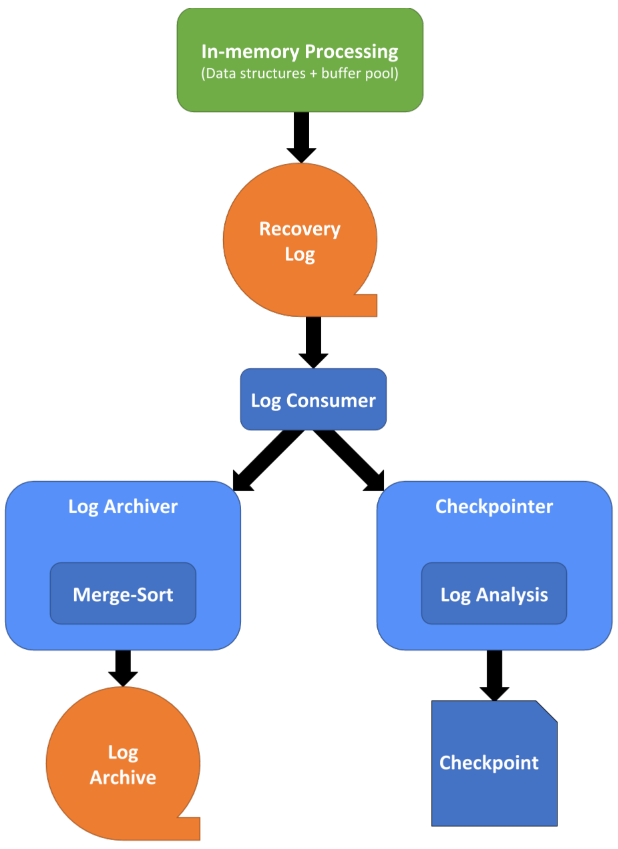


Figure 3.3: Producer-consumer abstraction.

Chapter 4

Decoupled Page Cleaner

4.1 Main Idea

As mentioned in Section 2.3, there are many different situations where flushing a database page from memory to persistent storage might be required. In any of these situations, the task of writing the pages is delegated to a page cleaner service. Considering a scenario where the whole working set fits in main memory, there are no misses in the buffer pool and 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 in order to reduce REDO time during system recovery in case of a failure. Hence, with the ultimate goal of reducing REDO-time, it is desirable for the cleaner service to be constantly active and keep as many clean pages as possible in the buffer pool.

The problem is that, in order to flush pages, the cleaner service requires direct access to the buffer pool data structure, as well as unnecessarily complicated latching schemes. If the page cleaner service is too aggressive, it might generate too much interference (mainly in 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.

A decoupled page cleaner achieves that by asynchronously replaying the generated log records to pages on persistent storage, without requiring any access to the page images in the buffer pool. In more details, the decoupled page cleaner fetches pages from the database device into its own cleaning buffer. Then, log records related to the pages in the cleaning buffer are replayed in order to bring them to a more recent state (not necessarily the most recent state). Once all the log records referring to these pages were replayed, they are flushed from the cleaning buffer back to persistent storage. In other words, the decoupled page cleaner fetches pages from persistent storage and executes REDO on them.

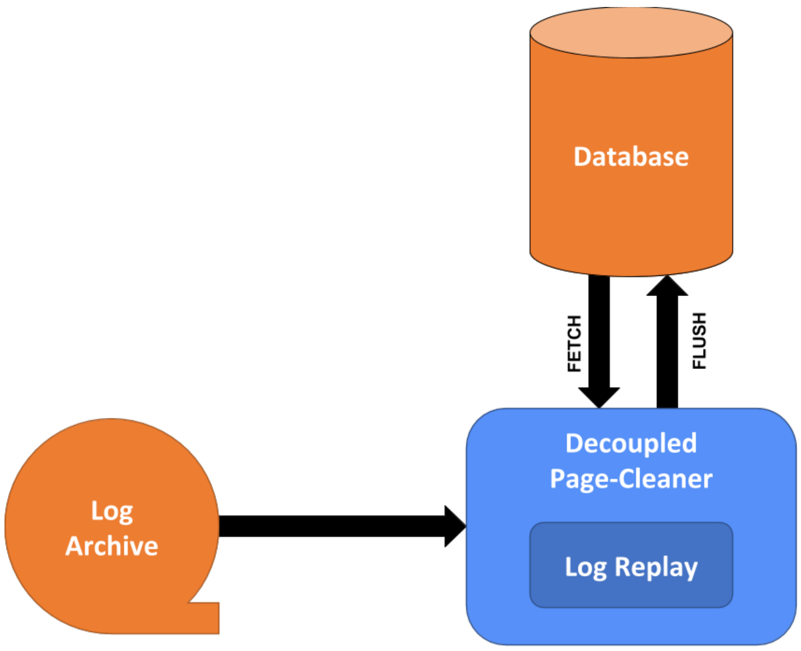


FIGURE 4.1: Decoupled cleaner

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 is not desirable for performance. 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, in order to avoid repeated work, the decoupled page cleaner keeps track of the LSN up to which pages were cleaned. Every time the cleaner is activated, it verifies if there is a new run file in the log archive whose log records are greater than the last cleaned LSN. If yes, the log records in the run file are replayed and the last cleaned LSN is updated. If not, there is no work to be done. In order to enable the decoupled page cleaner to resume from the "clean pass", the end LSN of the last completed replayed run file must be persistent as part of the system state. Figure 4.1 illustrate the main idea of the decoupled page cleaner based on log archive.

4.2 Implementation Details

The current implementation of the log archiver consumes log records from the recovery log device. In others words, log records have to be flushed to the recovery log device, consumed and sorted by the log archiver, and then flushed as a run to the archive device. Only then the decoupled-page cleaner is able to read the run file from the archive device to replay the log records. In a very unlikely scenario, which might happen only with a small buffer pool, this imposes a limitation worth mentioning.

If the buffer pool is completely full of dirty pages, the transaction processing might halt waiting for pages to be fetched from persistent storage. Since dirty pages must be cleaned, log records must be read from the archive device. It may happen that no new run was generated in the meantime and the log archiver is halted since no new log records are being generated. In such a case, the decoupled page cleaner might have to force a flush of the log buffer, wait for log records to be consumed and sorted by the log archiver and then flush the log archiver buffer to generate a new run. A way to avoid such behaviour is to enable log records currently being sorted by the log archiver to be read from memory, without the need of flushing them to a new run file. It is worth mentioning that this is a limitation of the current prototype implementation, and not a limitation of the design itself.

Two other issues must also be considered, since they comprise the idea of "decoupling". First, an efficient eviction policy should give preference for throwing clean pages away from the buffer in order to open space for new pages. Therefore, in order 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 in the 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.

In order to address both these limitations, the decoupled page cleaner can be combined with the idea of single-page recovery [5]. By employing single-page recovery, it is possible to determine if a page fetched from persistent storage is up-to-date and, if not, the restore process for that single page is triggered to bring the page to the most recent state before allowing further accesses to the page. That being said, the eviction policy can pick any page as victim and discard it without requiring a page flush, even if the page is dirty. In case the page was dirty, it is assumed that, by the time that same page is fetched again, the decoupled page cleaner will have already brought it to the most recent state. If the assumption does not hold, then the single-page recovery of that page is triggered. In other words, buffer algorithms do not have to keep track of dirty pages anymore. The idea of discarding a dirty page from buffer pool during eviction, without flushing it to persistent storage is referred as *write elision*.

Similarly to write elision, single-page recovery also enables *read elision* [6]. 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 otherwise 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.

Furthermore, considering the idea of recovery without UNDO [10], only log records of committed transactions are persisted. Consequently, the log archive will only contain page-update log records of committed transactions. Since the decoupled page cleaner replays the archived log records, it is possible to guarantee that, at any point in time, the persistent database is at a consistent committed state.

Finally, since all the decoupled page cleaner does is replay log records to persistent pages, it is possible to have the format of these persistent pages different from the format of in-memory pages. This introduces a new level of flexibility for the in-memory processing at a relatively low cost.

Chapter 5

Experiments and Results

5.1 Environment Setup

The main hypothesis to be tested by the following experiments is that a decoupled design can reduce the interference regarding main memory structures that might reduce the transaction throughput of the whole system. It is important to mention that performance improvements are not the main focus of the decoupled design, but to eliminate interactions in the code and dependency between system components. With that in mind, we implemented the algorithms previously described for decoupled checkpoint and decoupled page cleaner in the Zero storage manager[1] (based on Shore-MT [11]). In order to generate workload, we executed TPC-B benchmark using the Zapps package[2], which implements a stream of transactions that perform a large amount of small read-write transactions, such as debit and credit in 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. The 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 in performance. However, it is not a real-world scenario, since the recovery log must be stored in a non-volatile device.

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

It is important to mention that out system prototype does not implement write elision, meaning that the decoupled page cleaner still has to acquire latches on buffer pages for a short period of time, in order to mark those pages as clean when necessary. Furthermore, the decoupled page cleaner internal buffer is set to a size of 16 pages. The experiments with decoupled checkpoint were executed employing forward log scan and with log archiving disabled.

The transaction throughput in the results were measured by inspecting the log records generated by the execution of the benchmark.

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.

5.2 Checkpoint Experiments

The checkpoint experiments vary the frequency a checkpoint request is made. Taking checkpoints too frequently increases the interference in the in-memory data structure, which may harm the system transaction throughput. Figure 5.1 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 thousand transactions per second. Values on the x-axis represent how frequent a checkpoint request is made in milliseconds. Even though taking a checkpoint every millisecond is not a realistic scenario, it is done in order to stress the system and force as much interference as possible.

The less frequently a classical checkpoint is taken, the lower is the interference caused and, consequently, the higher is the transaction throughput.

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 in 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.

It is also possible to see that, taking classical checkpoints every interval higher than 1000ms does not produce a significant interference in the system and the throughput after this point equals to the one achieved by decoupled checkpoints.

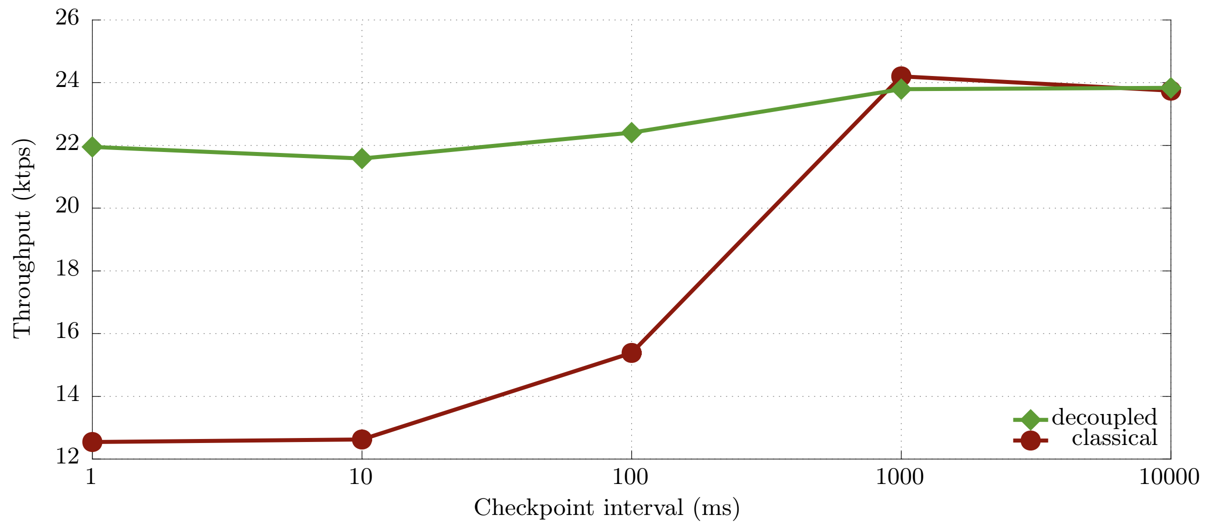


FIGURE 5.1: Throughput with log in SSD - Experiment 1

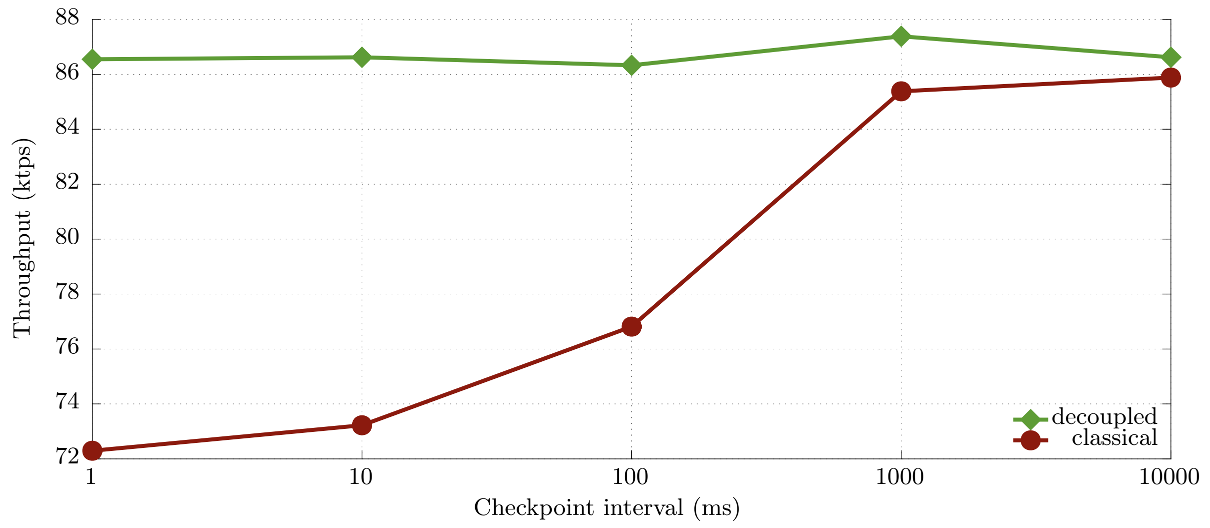


FIGURE 5.2: Throughput with log in MEM - Experiment 1

Figure 5.2 shows the same experiment, but having the log in a main memory file system in order to achieve higher transaction throughput. Again, the transaction throughput of decoupled checkpoints is higher when compared to classical checkpoints. We consider that decoupled checkpoints present a constant transaction throughput between different checkpoint frequencies and that any small variation is caused by random behaviours of the benchmark, since no I/O operation is required.

Figure 5.3 shows the CPU utilization observed when executing the benchmark with classical checkpoints and decoupled checkpoint, having the log in a SSD device. The values on the y-axis represent the CPU utilization, while the x-axis shows different executions with different checkpoint frequencies. The results are presented in the format of a box plot. The blue line in the middle indicates the median observed value. The box ranges from the 25-percentile on the lower boundary to the 75-percentile on the upper one. The lines extending below and above represent the minimum and maximum values, respectively. A box plot is used to provide a better summary of statistical distribution than the one provided by a single average value.

By analyzing the CPU utilization during the experiment, it is possible to see that the longer the interval between classical checkpoint requests are, the higher is the CPU utilization. The amount of CPU consumed for taking a classical checkpoint is assumed to be constant in all cases, since all a classical checkpoint has to do is iterate through the in-memory data structures. As seeing before, the longer the interval between classical checkpoints, also higher is the transaction throughput, which explains the higher CPU utilization. In other words, taking classical checkpoints less frequently allows the system to run transactions more freely and consequently more CPU is consumed. This could explain the small difference that exists when comparing the CPU utilization of classical checkpoints running every 1ms and 10000ms, for example.

A similar situation happens when considering the CPU utilization of the experiment running with decoupled checkpoints. The CPU consumed by taking a single decoupled checkpoint is proportional to the amount of log records required to be scanned which is directly related to how old is the previous checkpoint (and consequently to the checkpoint frequency). However, when considering the scenario of the whole benchmark, the amount of log records scanned for taking a decoupled checkpoint is the same, it does not matter the checkpoint frequency. In other words, the amount of CPU required for taking 10 decoupled checkpoints that scan 10 log records each is the same amount required for taking a single checkpoint that has to scan 100 log records.

As seen in Figure 5.1, the transaction throughput of classical checkpoint and decoupled checkpoint is the same when taking checkpoint in intervals larger than 1000ms. With that in mind, we can assume that the difference of CPU consumption in these cases indicates that the decoupled checkpoint consumes a small extra amount of CPU than the classical checkpoint.

Figure 5.4 shows the CPU utilization for the experiment run with the log in a main memory file system. Since there are no I/O operations required for committing transactions and, consequently the throughput is higher, there is more CPU consumption when compared to experiment with log in SSD device. Taking that into consideration, the results here are similar to the ones already mentioned.

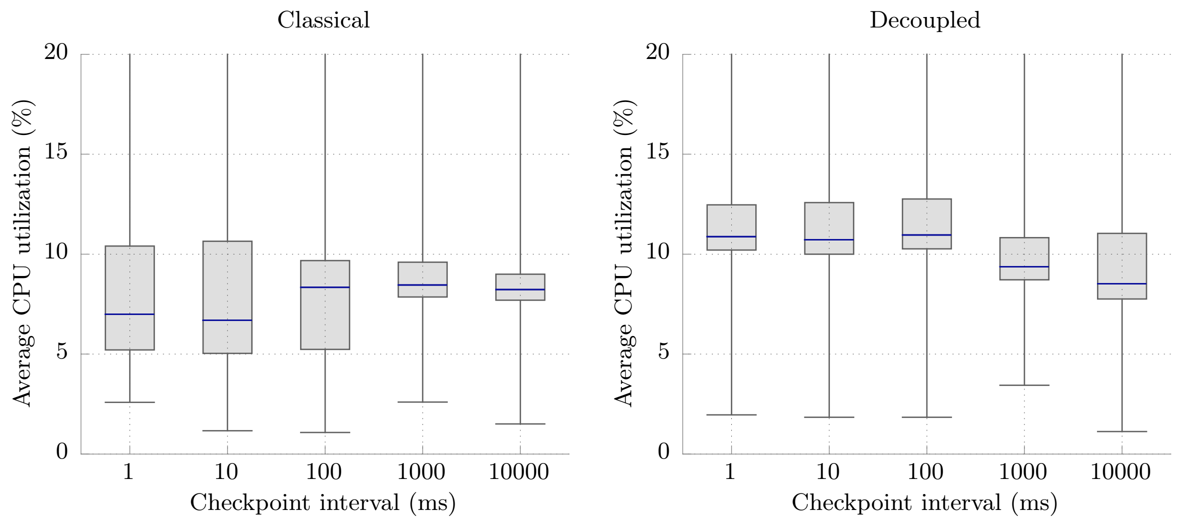


Figure 5.3: CPU with log in SSD - Experiment 1

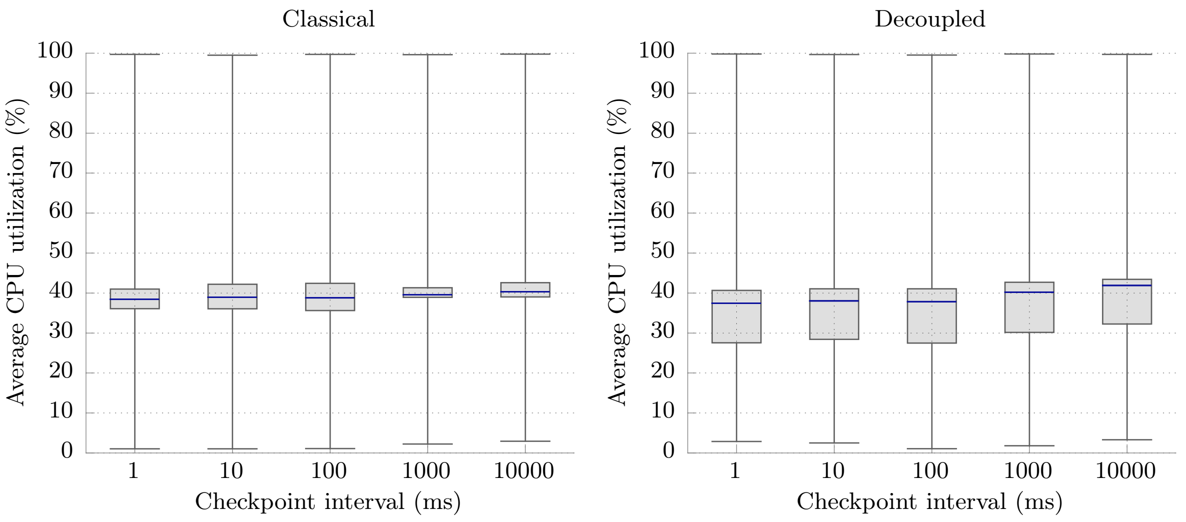


Figure 5.4: CPU with log in MEM - Experiment 1

5.3 Page Cleaner Experiments

Similarly, the page cleaner experiments are also made by varying the frequency the cleaner service is activated. The assumption was that activating the cleaner more frequently would induce a higher interference. However, Figure 5.5 shows that the transaction throughput varies very little along the different activation intervals. It is not clear if such behaviour is justified, since it is expected that activating the page cleaner every millisecond would generate much more interference, and consequently less throughput, than activating every 100 seconds. A deeper analysis of the page access pattern of the TPC-B benchmark is required for determining if there is a significant difference in the amount of pages dirtied for different cleaner activation frequencies.

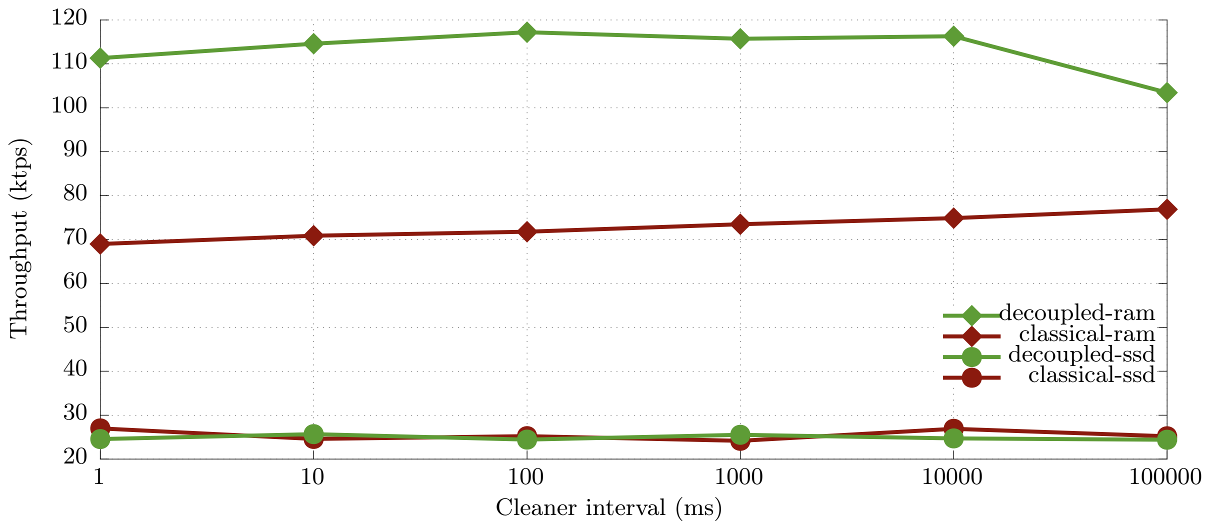


FIGURE 5.5: Cleaner Throughput

Nevertheless, it is possible to see that there is no significant improvement in the transaction throughput when running the experiment with log in a SSD device. However, the same experiment with log in a memory file system shows that the decoupled page cleaner achieves a much higher transaction throughput than the classical cleaner, since it avoids most part of the original interference. It is worth mentioning that, even with the log in main memory file system, the decoupled page cleaner still requires additional I/O operations for reading the log records from the log archive device (SSD).

Figure 5.6 and Figure 5.7 show the CPU utilization during the benchmark execution with the log in a SSD device and in a main memory file system, respectively. The CPU utilization is higher for the experiment with log in main memory due to the higher transaction throughput. Besides that, the experiment with classical and decoupled cleaner have similar CPU utilization. The classical page cleaner has the effort to sort the pages in order to enable sequential writes, while this happens implicitly in the decoupled cleaner, since the pages are read from the log archive already in a sorted order. On the other hand, the decoupled page cleaner requires additional CPU for replaying log records.

Finally, in Figure 5.8, it is possible to observe the write behaviour of the benchmark running with the log in SSD and in main memory. The cleaner activation frequency does not have much impact in the write bandwidth. In other words, the bandwidth is the same for writing 1 pages every millisecond and for writing 1000 pages every second. However, it is still possible to observe that most cases present a similar bandwidth with small variations, possibly limited by the device bandwidth. The exception is for the decoupled cleaner running with the log in main memory. A possible explanation is that, since there is a higher transaction throughput and more pages are dirtied, the decoupled cleaner does not take the most advantage of sequential writes, since it has an internal buffer of fixed size. Running experiments with different sizes for the

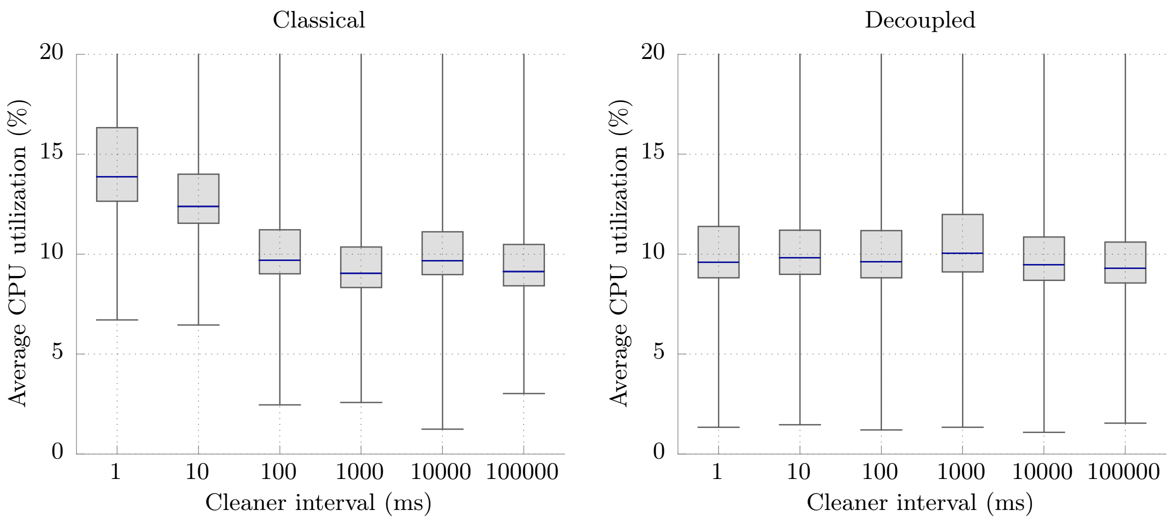


Figure 5.6: CPU with log in SSD

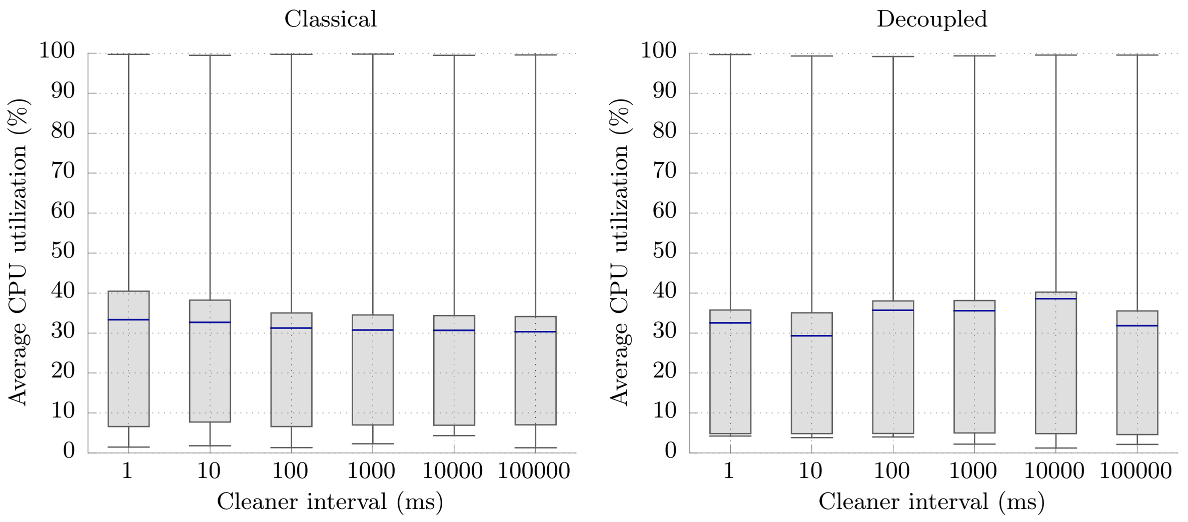


Figure 5.7: CPU with log in MEM

decoupled cleaner internal buffer could offer results to support this hypothesis. Nevertheless, a deeper analysis is also desirable to explain such behaviour.

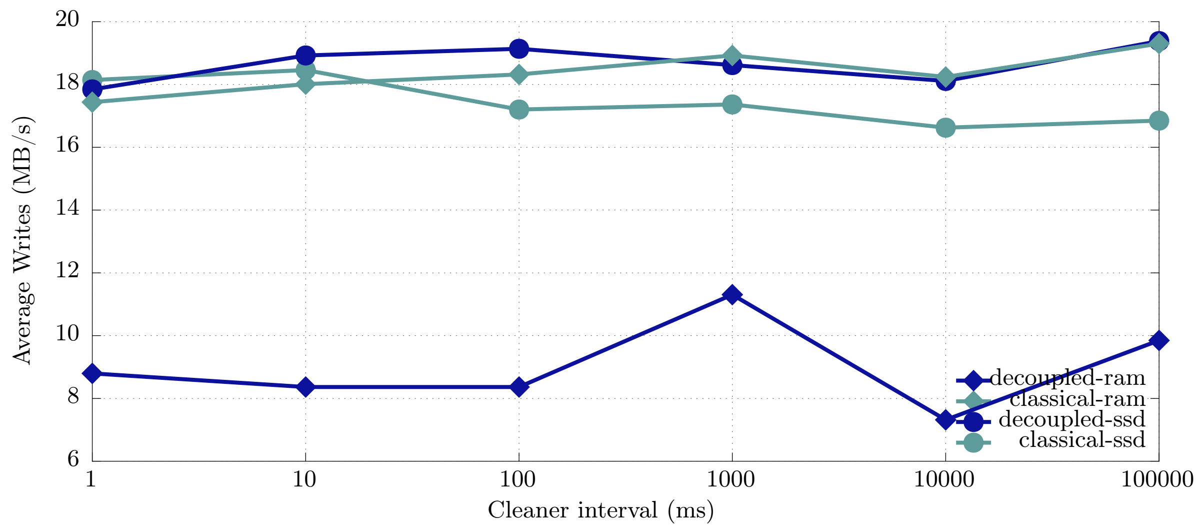


FIGURE 5.8: Cleaner IO

Chapter 6

Conclusion

This work proposed 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 on in-memory data structures, but also because it eliminates much of the code complexity introduced by the unnecessary interaction between components. More precisely, the goal of this thesis was to propose and evaluate novel checkpoint and page cleaner algorithms that are based on log information rather than on data collected from critical in-memory data structures. The original "coupled" system architecture can be seen in Figure 6.1.

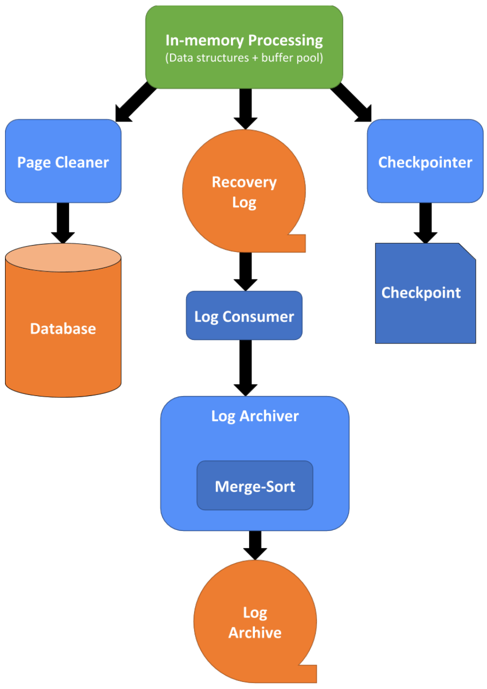


FIGURE 6.1: Original system architecture.

The original implementation for taking checkpoints required inspecting the buffer pool, the transaction manager, and the lock manager, in order to determine the system state. A checkpoint can be seen as the summary of the system state at a certain point in time, and it serves as means to reduce the length of log analysis during recovery. When a system failure occurs, the log analysis phase has to read the system state defined by the most recently completed checkpoint. After that, log records after the checkpoint are scanned in order to update the system state defined by the last checkpoint to the system state at the point of failure. The idea of a decoupled checkpoint is basically to employ the same logic used by log analysis for taking a checkpoint. Whenever a decoupled checkpoint is taken, the information from the last completed checkpoint is read and the following log records are scanned in order to bring the old system state to the current state. By doing so, not only we avoid latching the in-memory data structures for inspecting elements, but we also reduce the code complexity in the sense that these structures do not have to worry about a checkpoint thread querying its elements at arbitrary times.

In addition to reducing the log analysis length, it is desirable to reduce the REDO length by keeping a low number of dirty pages in the buffer pool. Most modern database systems implement a page cleaner service responsible for periodically flushing dirty pages from main memory to persistent storage. Similar to checkpoints, the page cleaner must access the buffer pool data structures to determine which pages should be cleaned. Moreover, the original implementation of the page cleaner introduces unnecessary code complexity by copying pages to be cleaned to an internal buffer, sorting the pages, and marking pages as clean after they are flushed. The proposed decoupled page cleaner avoids most of these interactions with the buffer pool required by the original page cleaner. By having a log archive partially sorted by page identifier and an efficient indexed access to the log records, the decoupled page cleaner works by simply fetching pages from persistent storage to an internal buffer, replaying log records to those pages and flushing them back to disk. Furthermore, not only it offers a less complex algorithm, but also an interesting set of features, such as working in synergy with the write elision and read elision techniques, which are enabled by single-page recovery.

The experiments presented in this work show that both decoupled strategies, for checkpoint and page cleaner, improve the transaction throughput of the system without implying significant additional costs. Results from a more realistic benchmark environment are desirable to verify the true impact of a decoupled architecture in the real world. Moreover, an in-depth analysis of the page cleaner behavior, e.g., with profiling tools, is required in order to highlight the true interference provoked by the page cleaner. Also, the implementation of write elision will provide a totally decoupled page cleaner and running experiments under this scenario should provide more satisfactory results in terms of performance.

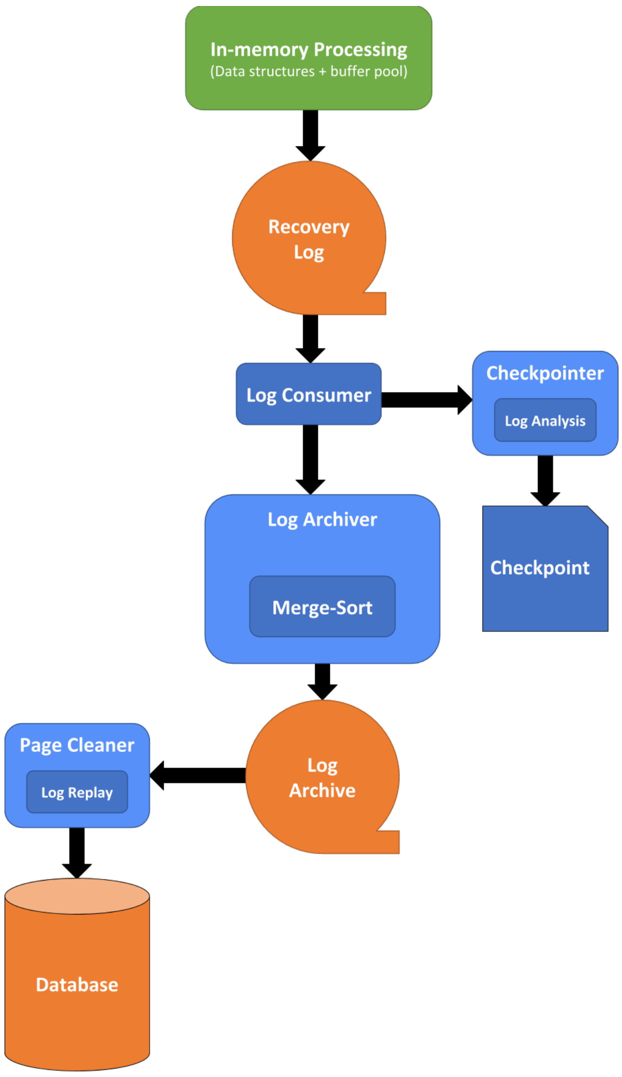


FIGURE 6.2: Decoupled system architecture.

Finally, as important as the performance improvement is the simpler and more modular system design achieved by the proposed decoupled architecture, as seen in Figure 6.2. By decoupling the design, not only the dependency between components is avoided, but the code becomes simpler in such a way that the intrinsic complexity of concurrent programming is alleviated in components that are crucial for system performance.

Bibliography

[1] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007. ISBN 978-1-59593-649-3. URL `http://dl.acm.org/citation.cfm?id=1325851.1325981`.

[2] 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, March 1992. ISSN 0362-5915. doi: 10.1145/128765.128770. URL `http://doi.acm.org/10.1145/128765.128770`.

[3] Caetano Sauer, Goetz Graefe, and Theo Härder. An empirical analysis of database recovery costs, 2014.

[4] Phillip Johnson. Let's Talk Data, Apr 2014. URL `http://letstalkdata.com/2014/04/falling-cost-of-memory-1957-to-present/`.

[5] Goetz Graefe and Harumi A. Kuno. Definition, detection, and recovery of single-page failures, a fourth class of database failures. *CoRR*, abs/1203.6404, 2012. URL `http://arxiv.org/abs/1203.6404`.

[6] Goetz Graefe, Wey Guy, and Caetano Sauer. *Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, and Media Restore*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2014. doi: 10.2200/S00617ED1V01Y201411DTM039. URL `http://dx.doi.org/10.2200/S00617ED1V01Y201411DTM039`.

[7] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983. ISSN 0360-0300. doi: 10.1145/289.291. URL `http://doi.acm.org/10.1145/289.291`.

[8] Vitor Uwe Reus. A study of i/o behavior in database systems, Semptember 2014.

[9] Caetano Sauer, Goetz Graefe, and Theo Härder. Single-pass restore after a media failure. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, pages 217–236, 2015. URL `http://subs.emis.de/LNI/Proceedings/Proceedings241/article11.html`.

[10] Caetano Sauer and Theo Härder. A novel recovery mechanism enabling fine-granularity locking and fast, redo-only recovery. *CoRR*, abs/1409.3682, 2014. URL `http://arxiv.org/abs/1409.3682`.

[11] 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. ISBN 978-1-60558-422-5. doi: 10.1145/1516360.1516365. URL `http://doi.acm.org/10.1145/1516360.1516365`.