TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Evaluation and Demonstration of Instant Recovery Techniques for Transactional Database Systems

by

Gilson Souza dos Santos

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

in the
Fachbereich Informatik
AG Datenbanken und Informationssysteme

April 2017

Declaration of Authorship

I, Gilson Souza dos Santos, declare that this thesis titled, 'Evaluation and Demonstration of Instant Recovery Techniques for Transactional Database Systems' 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: _____

"He who says he can and he who says he cannot are both usually right."

- Confucius

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Abstract

Fachbereich Informatik
AG Datenbanken und Informationssysteme

Master of Science

by Gilson Souza dos Santos

This thesis implements a demonstration platform to showcase database recovery techniques based on write-ahead logging. It focuses on modern instant recovery techniques that compete with traditional ARIES. On this platform, the database enthusiast is able to run different benchmarks in a prototype database system and observe, through charts and progress bars, how the system recovers from injected failures. The prototype system is able to recover from single-page failures by using Single-Page Recovery, media failures by using Instant Restore, and system failures by using either Instant Restart or ARIES restart. We describe how each of these techniques is implemented, their differences, advantages, and disadvantages.

In order to launch the recovery processes, we implemented the option to arbitrarily inject the three types of failures treated by the system. Each of these failures affects one or more internal components of the system. In order to analyze the internal state of these components, we embedded the server-side component of our platform in the database system runtime, thus providing efficient access to internal data structures, logged events, and execution traces.

The server-side component aggregates system information in statistical tables and makes them accessible through a REST API that delivers JSON data. We use this API in the development of a simple Web client that graphically displays these statistics. Furthermore, the client allows the injection of failures and reports on the progress of each recovery phase in real time.

Acknowledgements

I would like to express my gratitude to my advisor Caetano Sauer. He was not just my mentor during this work, but my guru during my entire master degree. All nights and weekends that we worked together during the last three years will always be gently carried in my memories.

I would like to thank Prof. Dr. Theo Härder for the opportunity to work in his research group and for all the assistance that he and his group gave to me. I was extremely fortunate to contemplate his impact in the life of many people and many projects during these years.

I am grateful for all the colleagues and friends that shared all bad and good moments during the past years. There are so many people, which makes hard to mention them all here. So I would like to specially mention Lucas Lersch, who shared not just the office with me but also his strange vision of the world, Claudio Busatto, who spent days and nights studying with me before exams and drinking beer when the exams were over, and Filipe Esperandio, who even living in Brazil was always present.

I would like to expose all my gratitude to my girlfriend Carmen. She was the one who brought light to my dark days and watched me close during all the process of this work.

Finally, I would like to offer my thanks to all my family and friends. I am particularly grateful to my mother, Rosangela, for raising me on her own and always helping me to achieve my dreams.

Contents

Dedicated to Rosangela

Chapter 1

Introduction

1.1 Motivation

During the last years, a set of new database recovery techniques, known as Instant Recovery [1], was developed in order to increase database availability in presence of failures. Such techniques differ from traditional ARIES [2] due to their on-demand approach, which enables the system to be available while database recovery is still in process. In order to demonstrate these new techniques and compare them with traditional ARIES, we developed an interactive tool, which allows the demo user to insert failures in the system and visualize the process of recovery of the DBMS through instantly generated graphics.

Besides the graphics, which track the recovery state, our demo tool also keeps track of several other system activities. By visualizing these activities (i.e., the internal system state) while the DBMS recovery process happens, the user can observe other related information, such as the fact that the system is available for new transactions earlier when it uses Instant Recovery instead of ARIES.

The differences observed when we compare both techniques occur due to the different approach used for each of them to recover from a failure. Although both techniques have similar steps and are based on Write-Ahead Logging, they have different strategies to execute their steps. As an example, both systems use three similar steps in restart recovery: Log Analysis, REDO pass, and UNDO pass. However, a system using ARIES is available for new transactions only when the first two steps are finished; meanwhile, a system using Instant Recovery would be available after the first step and would execute the second step on demand (i.e., when the system needs to access the data to be recovered). This Instant Recovery decision for executing the REDO pass concurrently with new transactions explains the reason for its earlier availability.

1.2 Contribution

The contribution of this work is the creation of a demonstration platform developed for inserting failures in the system and analysing the DBMS recovery and its internal state. Such analysis has been already done in other studies [1] , however, they were done with offline, post-execution analysis of the data; meanwhile, our analysis is done while the system is running. Although this approach can be interesting for our demo user, it does not show differences in the results already presented besides a performance loss (if running in the same environment) due the constant iteration with internal components from the DBMS while it runs. Our main contribution is to make available a tool which enables users to generate failures and see the system recovering from these failures with no need of advanced knowledge about system crash, media failure, and page failure, and how to analyze the system behavior while it recovers.

This demo platform started as a guided research project [3], which resulted in an interface that enables users to see the internal state of the system. We enhanced this platform by adding several benchmark options, insertions of different failures, and tracking of the system recovery process. The platform was created based on an architecture of two tiers: the Web server, which is coupled with the database, and a Web interface, which is used by the user to execute benchmarks, insert failures, and visualize the DBMS state through dashboards. These two tiers communicate with each other through a REST API available on the Web server. Figure 1.1 presents how these communications are established in our system and the user iteration with the system and the database.

As we needed to have control of the database internal components, we use the Zero transactional storage manager [4] as a library in our system. Zero is a fork of Shore-MT [5] and is available as open source. The approach of instantiating the database as a system object gives us more flexibility to collect internal information and insert system failures. In order to get the internal information from the system and create such failures, we enhanced the Zero prototype code with new functions and internal statistics. All enhancements in the Zero prototype, as well as the Web server and the Web interface developed, are available as open source.

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 recovery is discussed as well as relevant details about ARIES system recovery and how Instant Recovery implements the recovery techniques that can be visualized in our system: restart recovery, media recovery, and single-page recovery. Limitations and important aspects are also discussed. Chapter 3 describes in detail the implementation of the Web server and the changes in the DBMS to generate failures and statistics. Advantages and problems faced are as well discussed. Chapter 4 describes the interface implementation and its usability. In Chapter 5, we run through some tests to

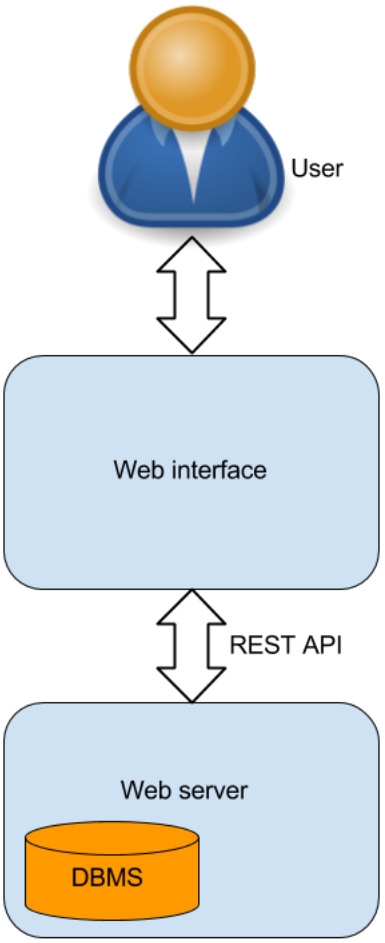

FIGURE 1.1: System communication

showcase the demonstration platform. Finally, Chapter 6 concludes this work by summarizing our contribution and providing guidelines for future work.

Chapter 2

Background Theory

2.1 Database Recovery

Database recovery is the process that guarantees that committed data will remain accessible and consistent in the case of a media failure or system crash. In other words, this process ensures the atomicity and durability properties of transactions (the "A" an "D" of ACID) [6]. Following a write-ahead logging approach, these properties are achieved by logging modifications to data cached in volatile memory (i.e., pages in the buffer pool) and by controlling their propagation to the persistent database.

Figure 2.1 presents the main data structures used by the DBMS in the propagation of modifications during update processing and recovery from failures. During update processing, transactions modify the database pages in the buffer pool and write log records to the recovery log. In case of a system failure, restart processing ensures up-to-date buffer pool contents from the available database and the recovery log, along with relevant server state in the transaction manager and the lock manager. Database backups provide long-term storage for database contents and the log archive provides long-term storage for log records. These forms of *archival* storage are used in restore processing to recover from a media failure [1].

There is a variety of system designs for a recovery component, and the ARIES mechanism [2] is the *de-facto* standard for disk-based architectures. ARIES has support for fine-granularity locking, does not require data pages to be flushed back to disk at commit time (no-force) and the pages can be flushed at arbitrary times, even if they contain uncommitted updates (steal). However, despite the several advantages of the ARIES design, recovery has a high impact on system availability, since the system can take hours or days to recover from a failure and accept post-failure transactions [1]. Instant recovery introduces some modifications to the ARIES design in order to reduce the time to accept new transactions and, therefore, increase system

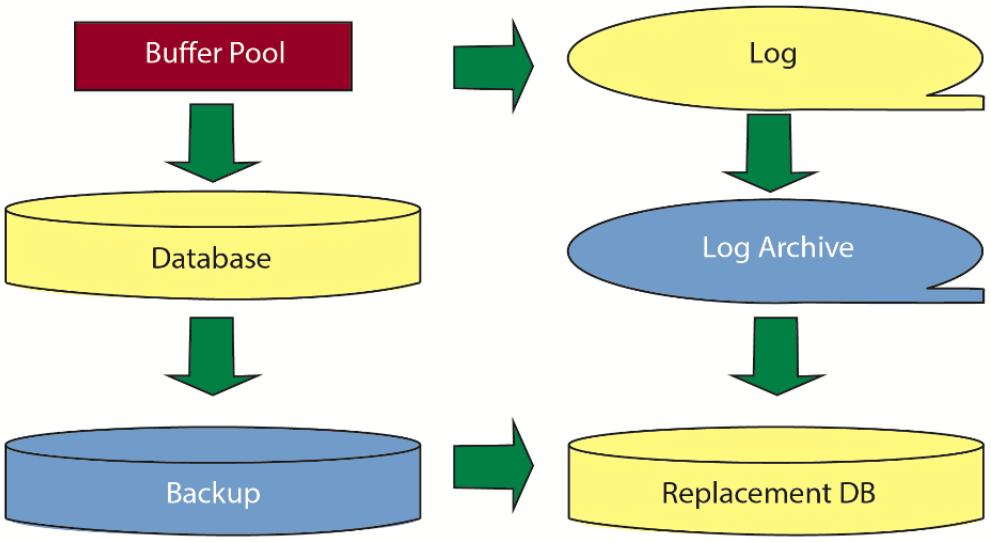

FIGURE 2.1: DBMS model and update propagation [1].

availability. Sections 2.2 and 2.3 describe both approaches and the methods used by each of them to recover from failures.

2.2 ARIES

ARIES recovery relies on the log created during normal processing, i.e., it uses write-ahead logging [2] to restore the system to its most-recent, transactionally consistent state. This log records all actions from transactions when there are changes to the recordable data records, ensuring that a committed action is reflected in the database, even with possible failures. The log is composed of multiple *log records*, which are the entities where the data changes are stored. Each log record has its unique *log sequence number* (LSN) assigned when it is appended to the log. This LSN is assigned in ascending sequence. This sequence is respected during the recovery process and it is used in order to determine if an operation was executed before another.

Every page in the database contains a field called PageLSN to track its state. Whenever a page is updated, its LSN field is also updated with the LSN of the log record that logged this update. If a failure occurs, it enables the system to determine which log records must be applied to the page to restore its consistent state.

When a transaction with changes to a specific page commits, this page, which resides in the buffer pool, is not necessarily written to disk at commit time. During this time the page on disk does not have the last modifications performed by the committed transaction and it is considered as dirty. The buffer pool manager registers the dirty pages in the dirty-page table,

which is used by the recovery system to determine pages in need of recovery. The entries in this table have two fields: PageID, which the identifier of the page, and RecLSN (recovery LSN), which is the LSN of the first log record that must be applied to recover the given page.

Each log record describing a transaction update has a pointer, which points to the previous log record created by the same transaction. By reading the previous log record that a transaction wrote, the DBMS can track all the changes performed by this transaction. This capability is important for the database to be able to rollback a transaction. When the database rolls back a transaction, it writes *compensation log records* to the log to track the operations undone. Every compensation log record has a field called *next undo LSN*, which guides rollback after an interruption, e.g., a system crash during a transaction rollback.

In order to avoid reading the entire log to restore the dirty-page table after a crash, the system periodically saves this table in a log record called checkpoint. This log record also contains a table with all active transactions in the system and the locks that they hold. During system recovery, it is necessary to reestablish the state of the system as it was immediately before the failure, and checkpoints are used as the starting point of this process.

2.2.1 Restart After a System Failure

When a crash in the DBMS occurs, all information on volatile memory is lost at restart, and the system recovery process kicks in. At this moment, the information contained in the log is used by the restart recovery in three steps: log analysis, REDO, and UNDO passes. After the conclusion of the second step, the system is already available for new transactions, provided that locks of active pre-failure transactions are re-acquired beforehand. The conclusion of the third step concludes the recovery process and brings the system to its last consistent state [2].

The first routine of restart recovery is log analysis. This routine receives as input the LSN of the master record, which contains a pointer to the last checkpoint taken before the failure. Log analysis recovers the dirty-page table and the transaction table from this checkpoint, and then it scans log records coming after this last checkpoint. While it reads these checkpoints, these two tables are updated accordingly. When a log record is found for a page that is not in the dirty-page table, a new entry in this table is created in order to include this new change. When a log record is found containing information that changes the state of a transaction, the transaction table is also modified to track this new state. Once this process finished, the system has the dirty-page table and the transaction table updated with the last state of the system and REDO pass starts.

REDO pass receives as input the minimum RecLSN (recovery LSN) from the dirty-page table produced during log analysis. Then it scans all log records from this point forward, verifying in

each log if the referenced page appears in the dirty-page table. If it does, it fetches the page in the buffer pool and applies the REDO action described in the log record if and only if its LSN is greater than the PageLSN value. No log records are written by this routine.

When REDO ends, the database is reestablished as of the time of system failure (e.g., *repeating history*[2]), except for updates performed by loser transactions that must be rolled back. The last step of restart recovery, the UNDO pass, solves this problem by rolling back the uncommitted transactions with compensating actions. The UNDO pass receives as input the transactions table produced during log analysis. The system recovers the last generated log record from each transaction in this table and uses it as a starting point to trace all changes performed by the transaction.

As already mentioned, the changes performed by the transaction can be tracked by following its last LSN, which points to the previous log record created by the transaction. By following these pointers, the system creates a chain of all log records created by the transaction. Based on this chain, the system identifies and rolls back all modifications of each transaction. When logged updates are undone, new log records are generated to store the information used to undo these changes. These log records are called compensation log records (CLR) and the pages affected by these log records have their PageLSN updated with the Compensation Log Rercord LSN. The compensation mechanism is required to enable record-level locking and protect against recurring system failures during recovery. It is essentially a form of logical UNDO, and is described in detail by Mohan et al. [2].

2.2.2 Database Backups and Log Archive

Backups are used to protect against data loss in case of media failures. There are three common forms of backups: a full backup saves all allocated database pages, a differential backup saves all database pages modified since the last full backup, and an incremental backup saves the last database pages modified since the previous backup of any kind [1].

These three types of database backups also differ in the data structures they require within the system architecture. A full backup requires only the data structures used for free space management. However, differential and incremental backups require additional bitmaps (one bit per database page) in order to identify pages which were previously saved in a backup. As differential backups save all data modified since the last full backup, their bitmaps can only be reset by full backups. The bitmaps for incremental backup are reset by any kind of backup [7].

While backups provide long-term storage of the database, the log archive provides long-term storage of the recovery log. As the log is typically held by latency-optimized devices (e.g. flash disks), there is a necessity to recycle space in these devices. Log archiving copies the log to

cheaper devices optimized for capacity and reliability and, during this process, may suppress all the information irrelevant for media recovery in order to save disk space.

2.2.3 Restore After a Media Failure

When a media failure happens, restore operations are executed, utilizing backups, log archive, and recovery log. Figure 2.2 presents the interaction among these components during this recovery process. In ARIES restore, the system first restores the most-recent full backup in a replacement disk that is formatted but empty. After that, all incremented backups created after the full backup are applied. Notice that Figure 2.2 shows only incremental backups since database administrators use either incremental or differential backups, but not both.

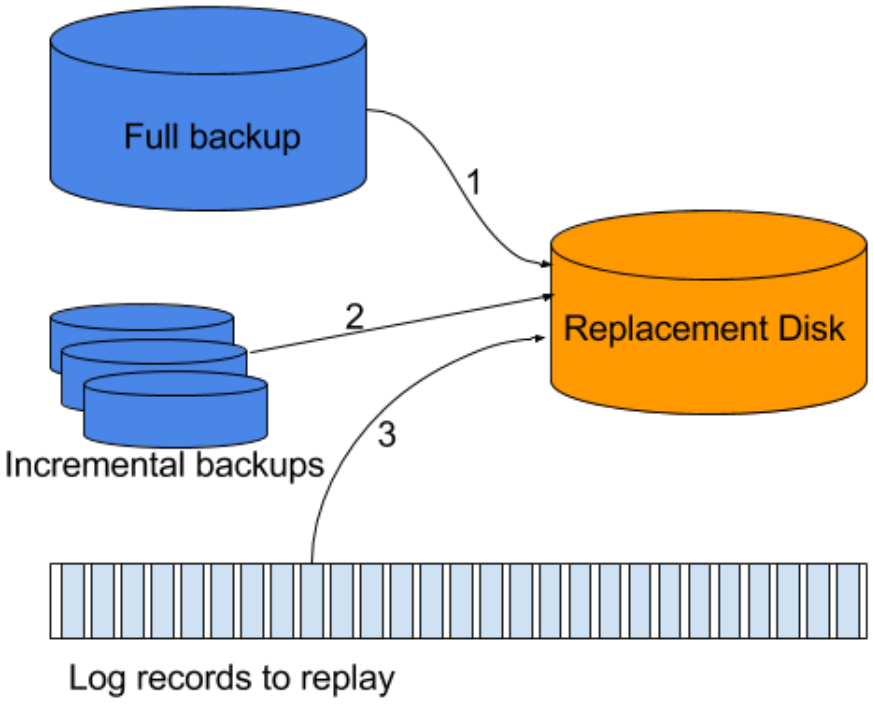

FIGURE 2.2: Media restore using the ARIES approach.

The last two steps of restore are based on the log. The system starts to replay all the log records created since the last backup (i.e., REDO pass). When REDO pass is completed, the system needs to roll back the incomplete transactions (i.e., UNDO pass). Due to the required I/O operation in the database, the REDO pass may take as long as the time that the original transactions took. For example, with daily backups, log replay can take as long as a day and UNDO operations may need to wait until REDO is complete.

2.3 Instant Recovery

Depending on the workload and the system environment, the recovery techniques used in ARIES may result in a system downtime of minutes or hours, since it needs to wait for the log analysis and REDO pass to complete before accepting new transactions. In these cases, Instant Recovery can reduce this time to seconds by recovering the system using an on-demand approach. By using this approach, the system is unavailable only during log analysis and both REDO and UNDO are deferred to on-demand execution.

Figure 2.3 presents an example of typical times spent in each step of restart recovery using ARIES. We can see the huge impact on availability, if the system allows new transactions direct after log analysis, since the log analysis is the fastest step from the restart recovery method, because it does not require random database reads and page updates.

FIGURE 2.3: Example of time spent on restart recovery [1].

Such an approach is possible thanks to new techniques that enable the system to perform recovery incrementally and on demand. Thus, Instant Recovery techniques do not need to complete the REDO pass to make the system available. The system can be opened to new transactions immediately after the log analysis phase and recover dirty pages on-demand. Associated with it, other strategies are used to make the recovery process faster. Instant restore uses indexed backups and log archives in order to replay the log sequentially, reducing random reads on the disk.

2.3.1 Single-Page Recovery

Recovery of individual pages on storage is not described by ARIES or any other traditional failure classes in transactional processing and database systems. Single-Page Recovery [8] addresses this issue by detecting the failure of an individual page, creating a per-page chain of log records to recover this page, and replaying these logs to this page.

In order to detect failures and identify the most-recent log record applied for each database page, the system has a *page recovery index* for each database or each table space. Each page of the database has an index entry, which points to the most-recent log record of the page. Each time the buffer pool writes a dirty page to storage, an entry in the page recovery index is updated with a new LSN value. Figure 2.4 shows an example of the system when the buffer

pool reads and writes the pages A and B. In the *page recovery index*, there are entries for both pages, which point to the last log record written in the log. By following the pointer *prevLSN* existent in each log record, the system can create a chain of log records specific for each of the pages.

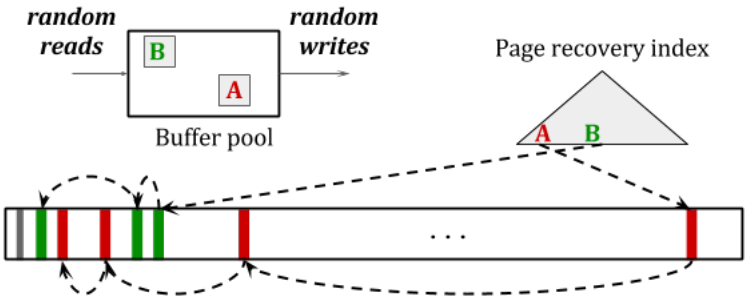

FIGURE 2.4: System example using a page recovery index to identify failures and create chain of log records.

If a transaction encounters a corrupt page, as in Figure 2.5, the system identifies the single-page failure, the log chain to this page is retrieved from the log and applied just to this specific page in the buffer pool before this page is available for the transaction.

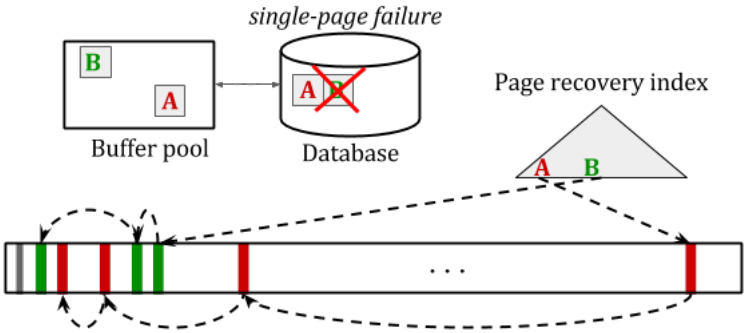

FIGURE 2.5: Example of single page recovery [9].

2.3.2 Instant Restart

Instant restart applies the same operations as in ARIES restart, but it enables incremental and on demand restoration in the REDO and UNDO phases. Like in ARIES, log analysis must collect from the log all information necessary to perform UNDO and REDO. It determines all pages that need to be redone and all transactions that must be undone. In the end of log analysis, all pages requiring REDO are registered in the buffer pool, which protects all REDO activity. Similarly, locks in the lock manager protect all UNDO activity [1].

After log analysis is finished, the system is available for new transactions. Figure 2.6 shows the state of the system at this point. If one of the new transactions requires access to a page registered in the buffer pool to be recovered, as the page "B" in the image, single-page REDO is called to recover this page. It checks the most-recent LSN in the dirty-page table generated by log analysis and applies the chain of log records for this page. After the page is redone, its reference is removed from the dirty-page table, avoiding future REDO for this page. Once this process is finished, the page is released to the transaction

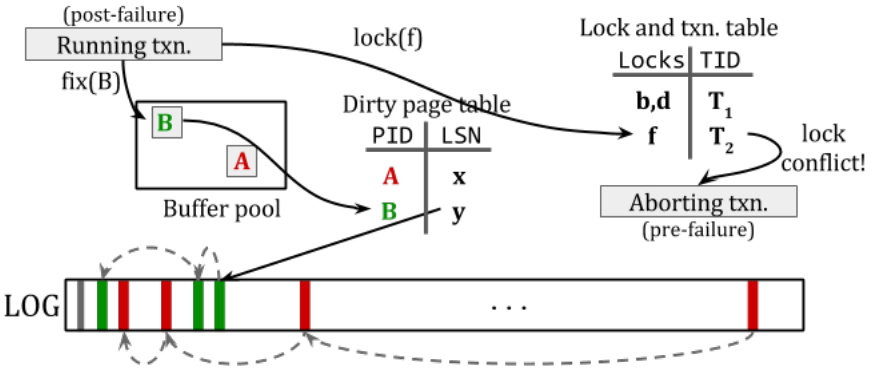

FIGURE 2.6: Execution of Instant Restart.

As on-demand REDO needs the chain of log records to apply to the pages that need to be redone, an efficient access to these log records is essential. To achieve acceptable performance, these log records must remain available, with fast access, preferably in main memory.

Some pages registered in the buffer pool may not be required by any post-failure transaction, and the buffer pool may decide to write these pages back to disk. When that happens, the buffer pool must find another page to replace or initiate single-page REDO recovery to recover this page before writing it [1].

During log analysis, all exclusive locks from transactions active at the time of failure are reacquired and their per-transaction log chains ensure their successful rollback. As Figure 2.6 shows, when a post-failure transaction tries to access a page previously locked by a lost transaction, a

lock conflict occurs. At this moment, on-demand UNDO is invoked and the pre-failure transaction is rolled back. If UNDO touches a page still in need of REDO recovery, the page is recovered as described above. This process continues until the end of the transaction rollback, when a commit log record is written and the acquired locks are released and subsequently granted to the new transaction. The process of undoing a log record applied to a page is the same as that described in ARIES, with compensation log records and UndoNext pointers.

2.3.3 Single-Pass Restore After a Media Failure

As ARIES Restore, Single-Pass Restore [10] is an offline recovery design to restore a media after a failure. Offline recovery is performed over a replacement media which is formatted and empty. The full backup and the incremental backups are restored in the replacement media and then log records are replayed to update the media to the database most-recent state before the crash and UNDO operations to rollback incomplete transactions. Offline recovery does not allow new transactions in the system to access the failed media, i.e., if the system is running, the new transactions can only access other volumes, which may exist in the system.

In the ARIES restore approach, after restoring the backups into the replacement media, the system starts to replay the log in the same order that the log was created. This process can take as much time as the system took to generate the log records. This happens because a number of random read operations executed on the disk. For every log record replayed, the system loads the page to be restored, replays the log, and writes the page back into the disk. Single-Pass Restore optimizes this process by using different strategies during log archiving.

In the Single-Pass Restore, the log archive is separated in runs sorted by volume and page identifiers such that, within each run or partition of the log archive, log records are clustered and sorted first by volume, then by page identifier, and finally by the original LSN of the log records.

When compared to a traditional, unsorted log archive, the crucial advantage of a partially sorted log archive is the efficiency of its use during a restore operation. Replaying the log records in an unsorted log archive requires many random accesses in the replacement database. In contrast, a single merge step can merge many runs from a partially sorted log archive.

Another strategy used by log archiving is to suppress and compress log records as much as possible. All log records not necessary to the restore process, are not archived, saving disk space and time.

Figure 2.7 presents the system during Single-Pass Restores, with a full backup, incremental backups, and log archiving runs partially sorted. The restore is executed page by page. After the system takes the page from the full backup, it applies the incremental backups to this page,

and replays all log records from all runs to the page. The page is written to the buffer pool, which writes it sequentially to the disk. As all logs were already replayed to this page, it will not be loaded and updated again until the end of the restore process, resulting in a much faster process than traditional ARIES restore.

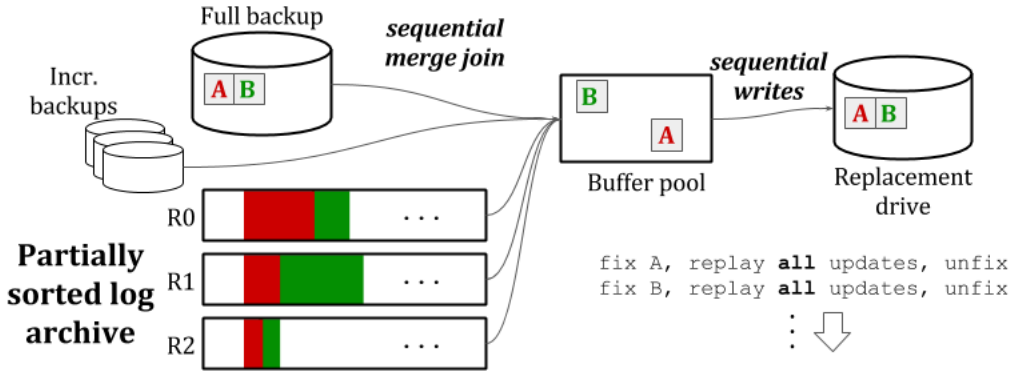

FIGURE 2.7: Single-Pass Restore [9].

2.3.4 Instant Restore After a Media Failure

In previous approaches of restore after a media failure, as the one presented in Section 2.2, the system needs to perform an offline restore of the media, i.e., the system is unavailable to new transactions until the media is recovered. Different from these approaches, Instant Restore [1] permits queries and updates basically immediately after a replacement media is available for recovery (i.e., formatted but empty). Besides the earlier availability of the system, this approach also improves considerably the time of restore when compared to ARIES.

The main technique of Instant Restore are on-demand restore operations, which are executed over segments (a page or a set of contiguous pages). If a transaction tries to access a segment which was not yet recovered, this particular segment is recovered from the most recent backup and the log archive. After that, the pages are available to the transaction. The process to recover the page is very similar to Single-Pass Restore, but instead of to recover an entire media, Single-Pass Recovery would be applied over segments on-demand.

As the system needs to be able to restore the segments fast enough to allow recovery on demand, as in Single-Pass Restore, the log archive is also partially sorted into runs. However, in order to access the logs records from the log archive faster and avoid several random reads, the full backup and the log archive are also indexed [9]. Thus, if a single device fails, recovery from the log archive is faster than with traditional log replay.

Figure 2.8 presents an example of Instant Restore. The full backup and all the log archive runs are sorted and indexed. When a segment is loaded into the buffer pool, the backup version of the page is loaded and the log records to this page are replayed. As they are indexed and sorted, the system reduces considerably the random access to the disk, which makes this process much faster than a traditional log replay. After the segment is recovered, the buffer pool will be responsible for writing it into the replacement drive.

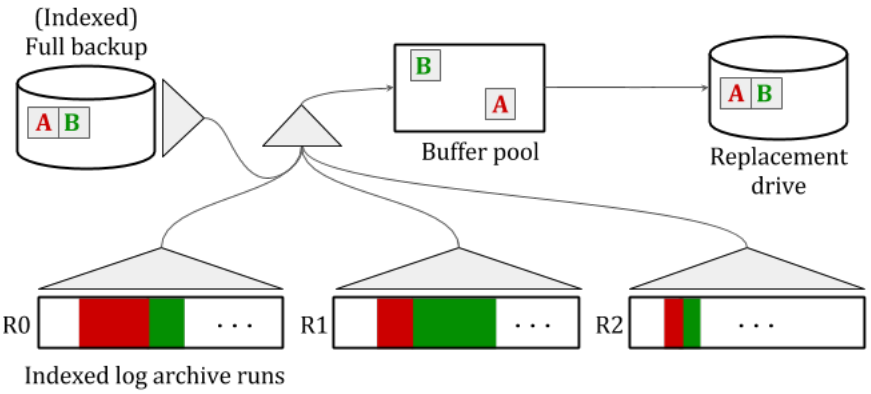

FIGURE 2.8: Execution of Instant Restore [9].

Chapter 3

Server-Side Implementation

3.1 Main Goals

As already mentioned, the demonstration platform is composed of a Web server and a Web interface. The platform has in the interface only the logic to send requests to the server and transform the server response into dashboards and status bars. All the communication is done through a REST API available in the Web server. This API can be used to insert failures, consult the state of the system, and start the benchmarks TPC-C [11] and TPC-B [12], which are implemented as C++ libraries in Zero.

Figure 3.1 shows how the server treats a call to start a benchmark or to generate a failure when the benchmark is already running. The benchmark is instantiated in the server, and its implementation gives access to the database environment. If the user inserts a failure, the server accesses the components present in Figure 3.1 through the benchmark object and generates the required failure. Currently, the server is able to insert three types of failures: system failure, media failure, and single-page failure. In case of a media failure, the system sets the volume as failed; if a single-page failure is inserted, the server takes a page from the buffer, changes its PageLSN, and writes it back to the disk; finally, for the case of a system failure, the program requests the buffer to stop any write operation in order to not allow the DBMS to write the changes to the disk and truncates the log in the current position to not allow any further progress by running transactions.

As the system recovers from a system failure during restart recovery, we are able to visualize and compare how ARIES and Instant Recovery behave in such scenario, since Zero has implementations of restart recovery for both techniques. When a media failure or a single-page failure occurs, we can just observe the behavior of instant recovery, since we did not implement the media recovery technique of ARIES and ARIES does not implement single-page recovery.

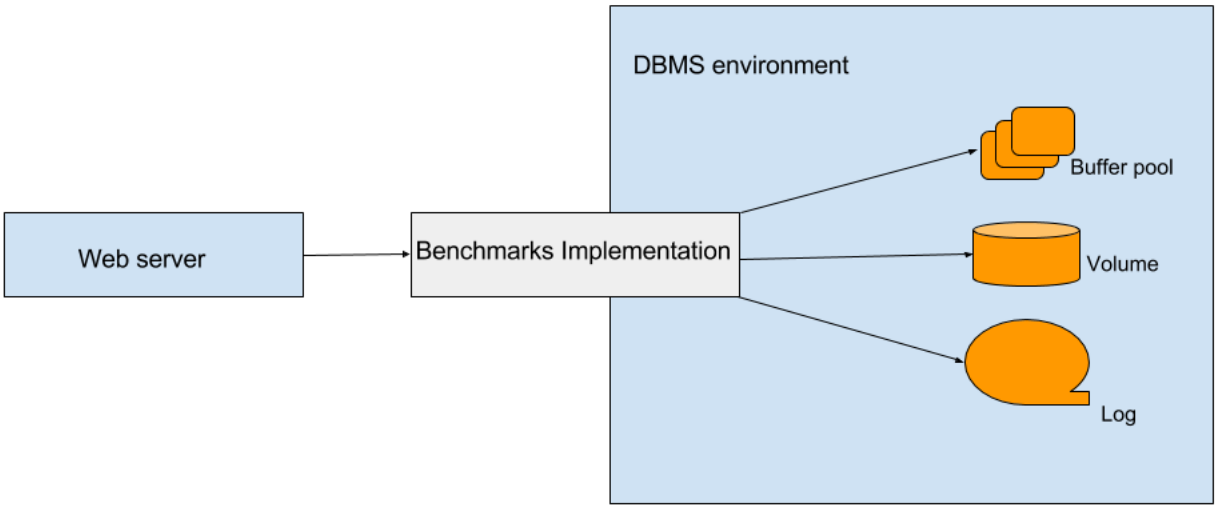

FIGURE 3.1: System architecture to start a benchmark or generate failures

When the system is running, the server collects statistics from the Zero storage manager using internal event counters as well as event log records processed from the log. These two sources of information are merged into a single statistics table and sent back to the interface in a JSON format [13]. We can visualize the described operations on Figure 3.2, which shows how our demo program extracts information about the executed operations in the DBMS.

In the following sections, we describe in more detail how each of the server characteristics was implemented, the necessary changes performed in Zero, and the challenges faced.

3.2 Implementing More Control Over Benchmarks

The benchmarks implemented in Zero run with a predefined configuration to stop. Currently, these three options are the duration of seconds, number of transactions to execute, and a log size. For some cases in our demo system, such measures are not necessary. For example, if the demo user runs the system with a predefined stop, the user needs to insert a failure in a point before the stop is achieved instead of deciding to insert the failure just based on the database

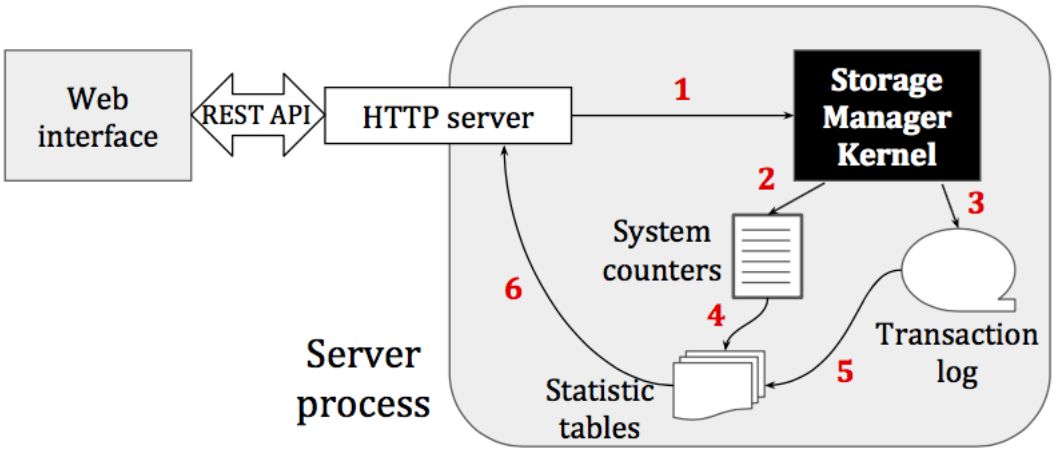

FIGURE 3.2: System architecture to collect statistics [14]

internal behavior. Based on that, we create a new option to run benchmarks indefinitely and only stop on request of the user.

By running the benchmark indefinitely, the user can observe statistics for the period of time that he wants and insert the failures at any point during this time. We also implemented an option to stop the benchmark at any point in time, which gives the user more control over the system. Such option is just available through the Web server since Zero does not provide any interaction with the user during its execution on the terminal.

3.3 Inserting Failures

In order to demonstrate the recovery process of Zero, we need to be able to insert failures in the system. As we demonstrate three cases of recovery, we had to insert a failure for each of them. Therefore, our system is able to insert three kinds of failures in the database system: system crash, which is used to showcase restart recovery; media failure, which starts the restore recovery; and page failure, which corrupts a page on storage so that it is eventually recovered by single-page recovery.

When a crash occurs, the system stops immediately. It is not able to finish its transactions nor create any other log record to be used during the restart recovery. Zero had already an implementation of such a failure, called dirty shutdown. This implementation stops the system during its process. However, the existing transactions were still able to abort and to write additional log records and other operations of the system may also write additional log records, as a checkpoint for instance. In order to insert a failure more similar to a real crash, we implemented a new shutdown mode, called *filthy* shutdown, which does not write any additional information for recovery.

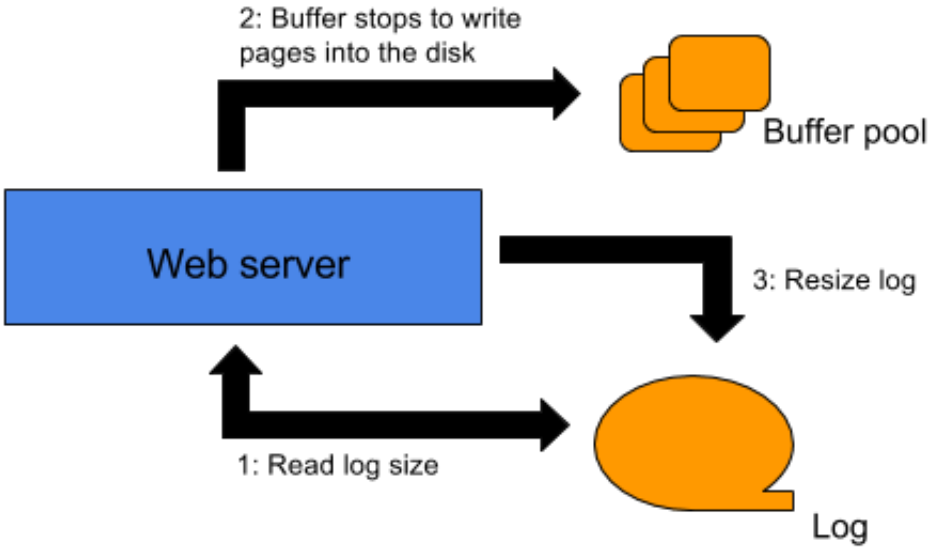

FIGURE 3.3: Implementation of shutdown filthy

Filthy shutdown implementation reads the size of the log in the moment that the routine is called. After that, it executes a call to the buffer manager requesting that it does not write any pages to the disk. Finally, before the shutdown process finishes, the log file is truncated to its size at the simulated crash moment. This truncation needs to occur because some other operations still write into the log, as the checkpoint, and the execution of such operations would hurt the simulation of a crash. As such writes are executed, we have the eventual problem of a new log file created during the process of shutdown. If such log is created, it is removed before we shut down the system. This pragmatic approach provides all mechanisms necessary to simulate a true system failure without the implementation efforts and subtleties involved in immediately stopping all system services and concurrent user transactions.

Figure 3.3, presents the iteration of the server with the components of the DBMS when the system performs the filthy shutdown. As the benchmark has access to the DBMS environment, all changes are executed through calls to the benchmark instance, however, the benchmark is hidden from the figure to facilitate its visualization.

The next failure implemented for the Web server was media failure. As Zero had already the implementation of a media failure, we just had to adapt it to the Web server code. The implementation receives a parameter informing a delay to start the media failure and, once this delay is reached, the system marks the media as failed, which starts media recovery. This delay parameter used is necessary for Zero for two reasons: first, since Zero was not implemented

as an iterative system, it can only execute operations defined before the benchmark starts. Therefore, the media failure could not be called during execution. Second, Zero is implemented for research. The time that the workload will run before the failure occurs is important for the researcher to be able to reproduce the exact environment used during the tests. However, in our demo platform, the use of this parameter is unnecessary, since we expect that the failure occurs moments after the demo user calls this failure. Therefore, when the server receives a request for a media failure, it just uses the Zero's crash implementation and sends zero as the delay parameter. The media is instantly marked as failed and the system starts to recover.

The demonstration of single-page recovery is the most challenging. As this recovery process occurs only if the workload accesses the page marked as corrupted, we had to be sure that the chosen page to insert the failure will be accessed. As we cannot be sure about it, since we cannot predict the workload behavior, our strategy was to choose a page present in the buffer, since there is a probability that the workload will access this page and, therefore, start the single-page recovery.

Once the failing page is chosen, we simply change its PageLSN value to zero and inform the buffer component to remove the page from the buffer pool. Before the buffer component removes the page from memory, it writes the page back to the disk, saving the changes that we made. When the page is stored on the disk, the benchmark may access this page, if it happens the system will verify the wrong PageLSN and will start the single-page recovery. However, as previously stated, we cannot predict when this process will occur and it may never occur.

3.4 Extracting the Recovery Status

The server presents in its API an operation to consult the state of the recovery processes. There are five tracked activities: log analysis, REDO pass, UNDO pass, media recovery, and single pages recovered. The first three are related to the restart recovery, and the last one shows the pages recovered due to the insertion of a single-page failure.

As log analysis reads a determined number of log records, we would like to present its progress, by informing the percentage of log records scanned at the moment that the user consults the system. However, due its fast process, we opted for presenting only if this process has finished or not.

When log analysis is completed, we have a table containing a list of all dirty pages and all lost transactions. In order to present the REDO pass progress, we consult the number of dirty pages on this table and start to track, with a counter, every page recovered by single-page recovery, since this is the process used to recover each dirty page. Such an approach can lead to an imprecise result because, if other failures occur and other pages are recovered during this

process, they would be counted as part of the REDO progress since both cases are treated by single-page recovery. We assume that such a case does not happen or does not have a big impact on the progress status.

In order to present the UNDO progress, we consult the number of loser transactions and the number of active transactions in the system. This approach is used to verify if log analysis has restored all the loser transactions in order to UNDO the operations. Once these transactions work are restored in the system, they behave as normal transactions, and we do not track their behavior.

The single-page recovery metric is a counter informing how many single pages were recovered. This counter is the same used during the computation of REDO progress minus the number of pages in the table generated by log analysis. This approach leads to the same imprecision of the REDO metric. If a new dirty page is recovered while REDO is still running, it would not count it as a new dirty page recovered, but as part of the REDO progress.

The last recovery mode is media recovery. We track media recovery progress by counting how many pages were recovered by the segment restoration procedure. The counter is incremented by the number of pages existing in the recovered segment and then we divide this value by the number of pages existing in the volume when it failed, which results in a percentage of pages recovered.

3.5 Gathering Statistics

Besides the recovery state of the system, the server also displays information about other internal operations being executed in the database. This information is collected through event counters and by reading log records and identifying the operations that generated them. The implementation of this process is presented here in order to cover completely the system and its functions.

The event counters are variables that are incremented when a certain operation occurs. By counting how many times an operation is performed, we can see patterns during the execution of the benchmark and compare the performance of the system in different situations, as using different recovery techniques, for instance. To illustrate how counters are produced and collected, we take an example scenario using the buffer manager component.

Inside the buffer manager component, there is a process called cleaner, which reads all pages from memory and writes the dirty pages to disk. If two or more pages in the memory can be written in sequence, the cleaner process can write these pages with a single I/O operation; otherwise, it needs to perform two or more accesses, which increase the write cost. Every time a

write operation is executed, the event counter that tracks this operation is incremented. When the benchmark is finished, we can check how many write operations the cleaner has executed and which one of them was the most frequent.

In our studies, the most interesting counters are the ones concerned with transaction manager activities, since they show when the system becomes ready to accept new transactions and how transaction throughput develops while the system is still recovering. In order to track these numbers, Zero has counters to track activities from the Transaction manager, such as new transactions, committed transactions, and aborted transactions.

The other approach used to collect information from the DBMS is by analyzing the log file that it produces. Log records stored in the log file contain information such as checkpoints, page changes, compensations, etc. We can read this information to determine when operations were executed and how many of them were performed. However, besides these log records already described, Zero also creates log records to track the start and end points of certain system activities, such as checkpoints, the phases of recovery, or the write of a dirty page.

In order to grab the information contained in the log, we use a tool that reads the log records and analyzes them. This analysis tool reads log record after log record in order to create a table, which presents the logged operations performed. This table contains a list of different types of operations performed and how many times they were performed during the execution of the workload.

In order to identify the events in the recovery log, the tool uses a field, which is presented in every log record and identifies the type of log record, and by analyzing this field, the tool can identify the operation that has generated this log record. As an illustration of this scenario, imagine that the tool reads a log record and verifies that its log type is the commit of a transaction. At this point, the tool knows that a transaction has committed and the table can be updated by incrementing the position, which refers to committed transaction operations.

As we were also interested in the timeline of performed operations, a special type of log record was introduced to determine the span of every second of the database execution. The system creates a log record of this type every second with the only goal of informing the tool that another second has passed since the last time a log record from this type was created. When the tool finds this log record, it stops to count operations in the current column of the table and starts to increment the next column, which results in a table representing the log and grouping logged operations executed every second. Figure 3.4 illustrates the creation of this table.

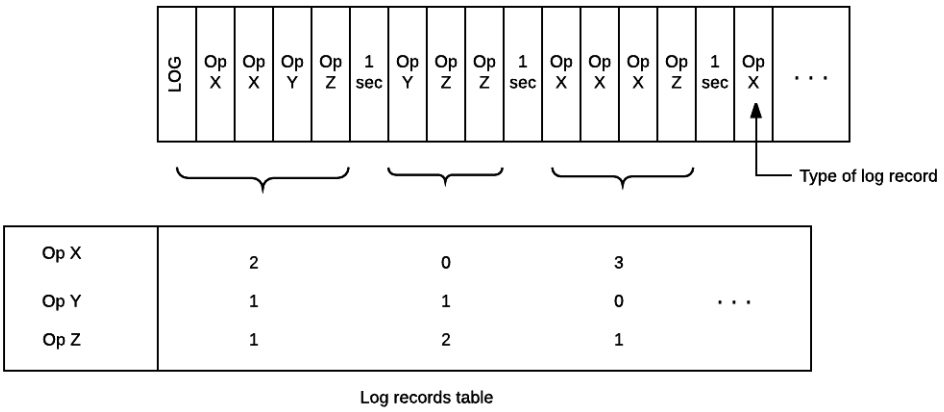

FIGURE 3.4: Creating a table to represent the logged operations

3.6 Counters Result Table

After gathering the data from the log and the list of counters updated in memory, we join these two sources in one table, so clients of the Web server do not have to make more than one request to query the internal state of the system.

The two sources of the database state are represented in different models: one as a table with a timeline showing the number of operations performed every second and the other only as a list containing the total number of events, without any notion of time. In order to join both sources, we had to decide which format would be more appropriated for a final table. As the notion of time was important to us, we decided in favor of the option with a timeline of performed operations. This decision forced us to present the counters existing in memory in the same way as the table based on the log (i.e. separated by seconds).

In order to implement the notion of time for event counters, we created a periodic process that reads the counters from the memory every second and inserts this information into a table, which has the same format of the table of logged operations. After every second, a new row in this table is created.

When a client requests the internal state of the database, the system joins the table created by our new process with the table created based on the log and creates a final table with both sources of information. This table is then converted into the JSON format, which makes it easy for external programs to use this data, since there are several frameworks designed to work with JSON. Finally, the Web server returns this table to the client.

TABLE 3.1: API routines

Name	Description	Parameters
`start_kits`	Starts the database and runs one of the two benchmarks available in Zero: TPC-B and TPC-C.	Benchmark; Load database; No stop; Duration; Transaction; Number of threads.
`is_kits_running`	Returns the state of the database, i.e., whether it is running or not.	No parameters.
`counters`	Returns all counters in the system, from both the log and system events.	No parameters.
`get_stats`	Returns counters collected from system events.	No parameters.
`agg_log`	Returns counters collected from the log.	No parameters.
`crash`	Inserts a system failure.	No parameters.
`mediafailure`	Inserts a media failure.	No parameters.
`singlepagefailure`	Inserts a single-page failure.	No parameters.
`recoveryprogress`	Verifies the state of log analysis, REDO Pass, UNDO pass, media recovery, and single-page recovery.	No parameters.

3.7 REST API

Our platform makes the JSON response available through a Web server, which also accepts a command to initiate the benchmarks. This Web server was built during the mentioned guided research using the *Boost.asio* library [15], and it is built in Zero in order to run the benchmarks and collect the database state. The server had already five routines presented in its API. We changed the start_kits routine, in order to send more parameters through post calls, and we added four additional routines, which are responsible for inserting failures and showing the progress of instant recovery. Table 3.1 presents the API routines, together with their descriptions and parameters.

Chapter 4

Web Client Implementation

4.1 Goals

The main goal of the interface is to facilitate the interaction between the user and the server. We achieve that by creating a Web interface that enables the user to inject failures in the DBMS by clicking on buttons and visualize the recovery process and the server-internal state through simple graphics and progress bars. This interface can run in any simple Web server, and it does not need to run on the same machine of the Database used to insert failures and observe the recovery process.

The communication between the interface and the database happens through the REST API available in the Web server. We tried to show the progress of the recovery and the state of the other operations as close as possible to the moment that the operation is happening, however, as each call to the database requires access to variables used by the system, if we execute too many of these calls in a small period of time, the database performance is negatively affected. Therefore, we execute these calls to the server every three (for recovery state) and five (for the other operations) seconds. Such an approach results in a delay in our interface comparing to the event counters in the database. This delay is considered acceptable, since our goal is to demonstrate the progress and, even with the delay, this goal is achieved.

4.2 Architecture

The base architecture of the interface was developed is described here in order to create a better understanding of the system and of our enhancements.

The architecture of the Web interface is based on Model-View-Controller (MVC) [16] pattern, which creates a clear division among the data (i.e. model), the interface presented to the user

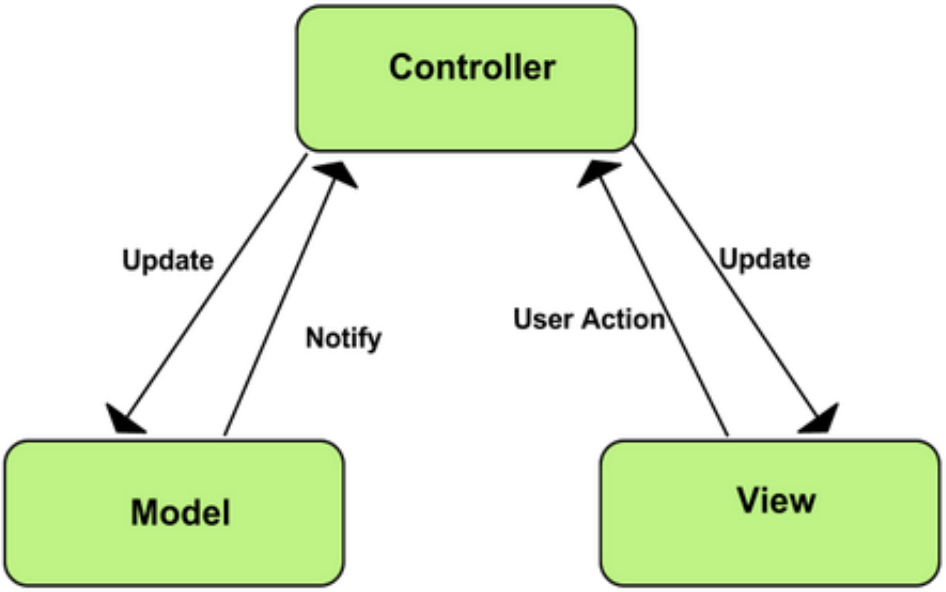

FIGURE 4.1: Model-View-Controller pattern.

(i.e. view), and the code that controls the interface and updates the model. Figure 4.1 shows the interaction among these components.

The Model represents the information (the data) of the application and the business rules used to manipulate the data. In our implementation, we have the counters and the progress of the recovery processes as Models, which store the data recovered from the REST API and information regarding its presentation in the interface, such as the counters currently selected for presentation, their names, and the axis in which to plot the counter values (right or left Y-axis). The View corresponds to elements of the user interface such as text, checkbox items, graphics, progress bars, and other elements presented in the Web page. Finally, the Controller manages the communication between Model and View. It handles user actions such as keystrokes and mouse clicks and takes the appropriated actions by updating the interface or the models. It also manages the communication with the Web server, sending the user request via the REST API and reading the response. Once the controller gets the response, it feeds it into the models existing in the system, updating the dashboards.

4.3 Interface and its Options

The interface was created using frameworks that facilitate the design and functions implementation. The charts are generated using the JavaScript version of Plotly [17], which is a framework to create charts, and the design is based on Bootstrap [18], which is an HTML, CSS, and

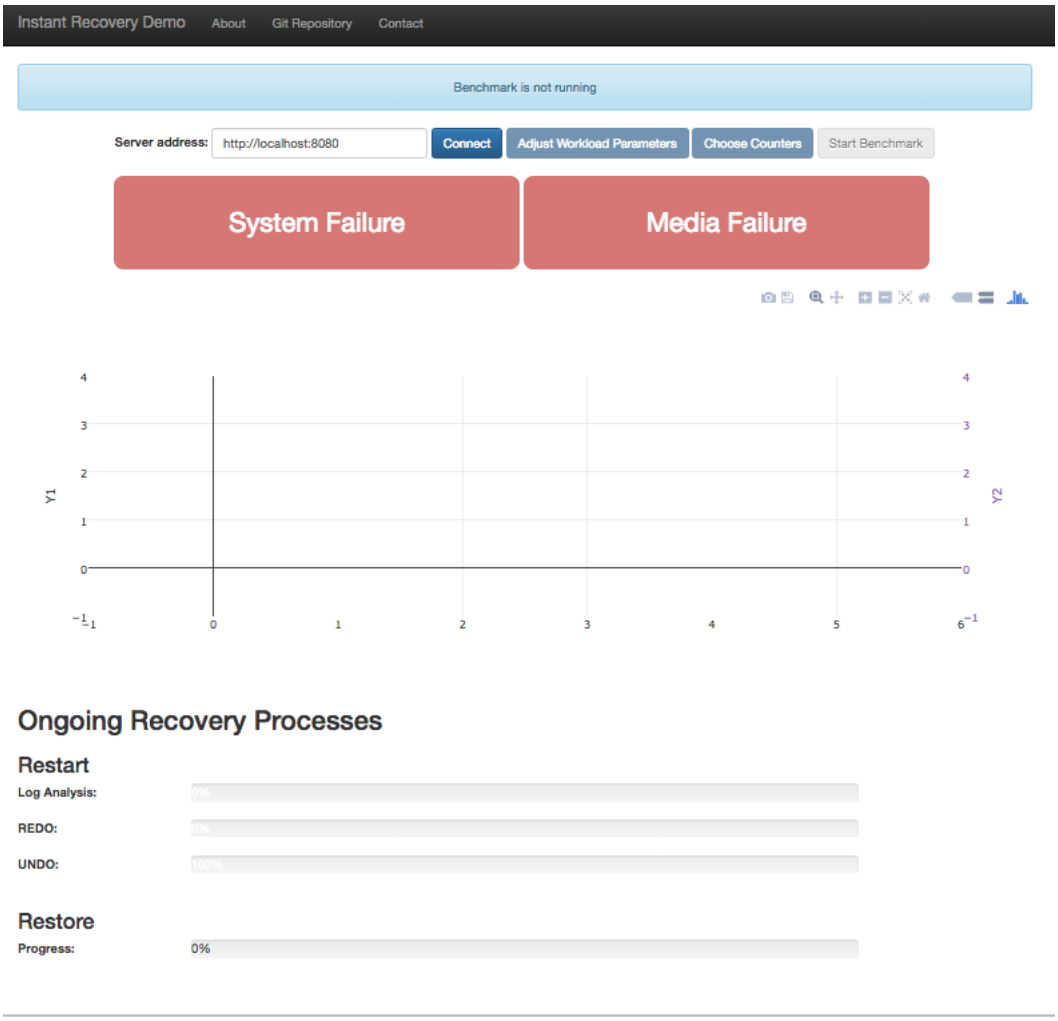

FIGURE 4.2: Interface of the system.

JavaScript framework to create Web pages. This framework adapts the interface to different devices such as desktops, tablets, and smartphones.

Figure 4.2 shows a screenshot of our demo interface before a benchmark starts. At the top of the interface, we can see the status of the benchmark, i.e., whether it is idle, initializing, or running. Below it, we have the dashboard to the user interaction with the system. The first field on the left of the screen is used to inform the server IP address, which runs our Web server with the database. On the right of it, there is a *connect button*, which connects the interface with the server and enables the other buttons in the screen. Then we have the buttons to adjust the benchmark parameters, the counters to visualize (which can be modified while the benchmark runs). Right below the controller buttons, there are two buttons used to inject two types of failures in the DMBS.

Below the dashboard of interaction, we present the counters on a dynamic chart. As different operations can be executed with higher frequency than others, we present the option to see

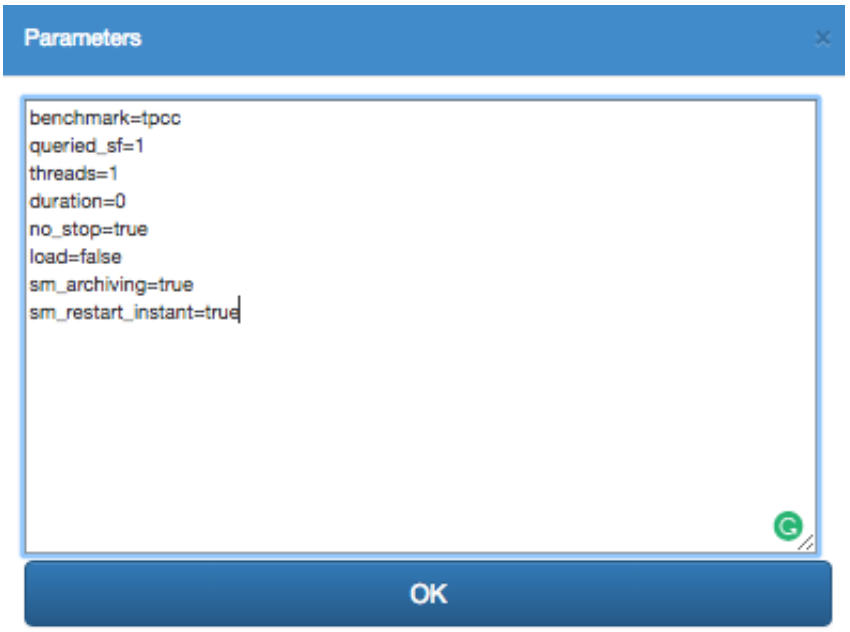

FIGURE 4.3: Options to run the workloads.

them in two different Y-axes. We group the ones with a lower frequency of execution in Y1 and the others in Y2.

After the chart of counters, we have the four bars to show the on-going recovery process. First the three bars on the instant restart section, which shows the restart recovery, and then the fourth bar on the instant restore section, which shows the progress of media recovery together with a number of pages recovered.

The *Adjust Workload Parameters* button opens the pop-up menu presented on Figure 4.3. On this menu, we are able to use the same options present in *Zero*. If we want to run the benchmark indefinitely, we set the option *no_stop* to true, if we want to run the benchmark during a certain period of time, we set the option *duration*, and so on.

By clicking on the *Choose counters* button, a pop-up menu with (currently) 276 event counters is presented. The selected counters will be displayed on the chart and can be changed during the execution. Figure 4.4 shows this pop-up menu and some of the existing counters.

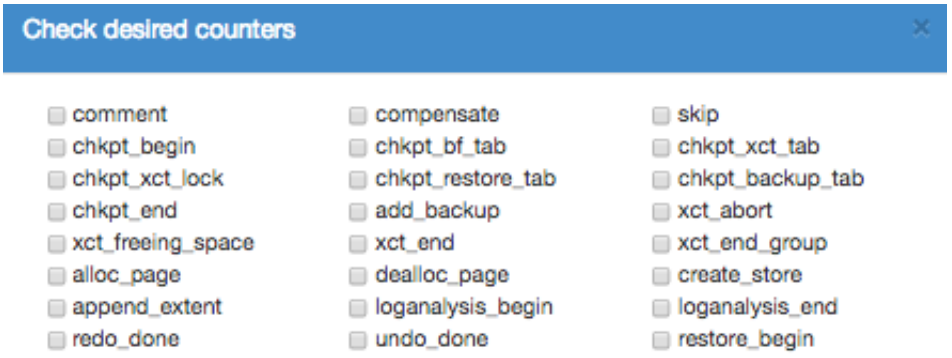

FIGURE 4.4: Pop-up to select the counters.

Chapter 5

Injecting Failures and Observing Recovery Processes

5.1 Environment Setup

As already mentioned, the demo platform can inject three types of failures. However, due limitations of our platform, we can just present the recovery of a crash for ARIES and Instant Recovery and the recovery of a media failure for Instant Recovery. The recovery of single-page failures is supported by the system, but it was removed from the Web interface because there is no interesting visualization for that process. Instead, a status message is printed at the console of the server component.

We executed the tests using the TPC-C benchmark with 10 worker threads and scale factor 10 in such a way that each hardware thread executes transactions referring to a single warehouse of the benchmark. Before injecting a system failure, the database is loaded and the benchmark is executed for two minutes, as presented in Figure 5.1. Such time was enough to generate dirty pages to be recovered by Instant and ARIES restart.

The server used to run the benchmarks has two Intel Xeon E5420 processors, each with 4 out-of-order cores running at 2.5 GHz and having a cache size of 6 MB. The server has 32 GB of RAM. The operating system running on the server is Ubuntu 15.04.

5.2 Injecting System Failures and Instant Restart Processes

After the system loads the database and runs the TPC-C benchmark for 2 minutes, we clicked on the `System Failure` button, which caused a crash, as presented in Figure 5.1. As shown

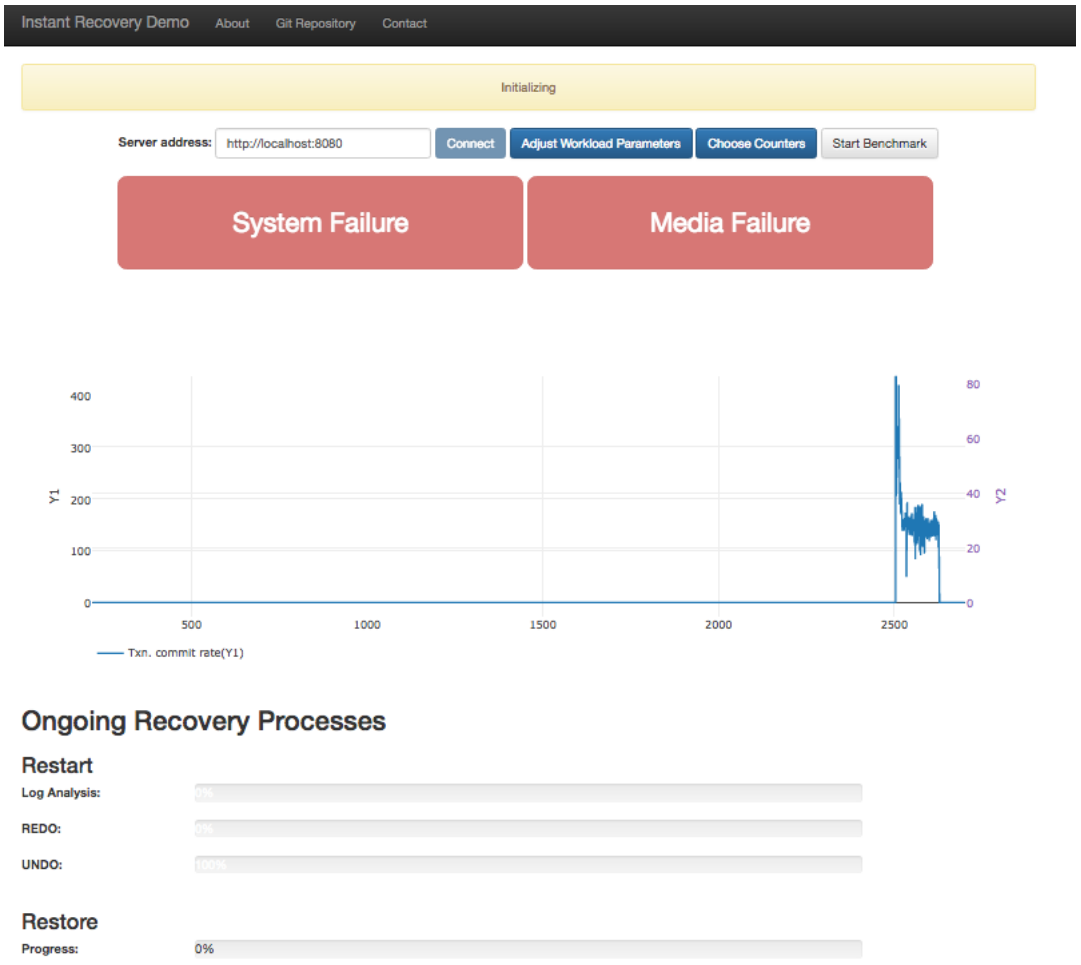

FIGURE 5.1: Loading database and injecting system failure.

in Figure 5.2, after approximately 180 seconds, the system starts to commit transactions even with the REDO process running, as we can observe on the progress bars at the bottom of the image. We also observe that, as the recovery process progresses, the system is able to commit more transactions per second.

In Figure 5.3, the restart process is completed and the system achieves the top of its performance after approximately 10 minutes running.

5.3 Injecting System Failures and ARIES Restart Processes

The next experiment uses the same setup as the previous one, but this time with ARIES restart. We can observe in Figure 5.4 that after approximately 500 seconds, the system started to commit transactions and directly achieves the highest performance, since the REDO pass has already executed and the buffer pool is already warmed up. The difference is 5 minutes and 20 seconds longer than Instant Restart took to accept new transactions, which is a big difference in this

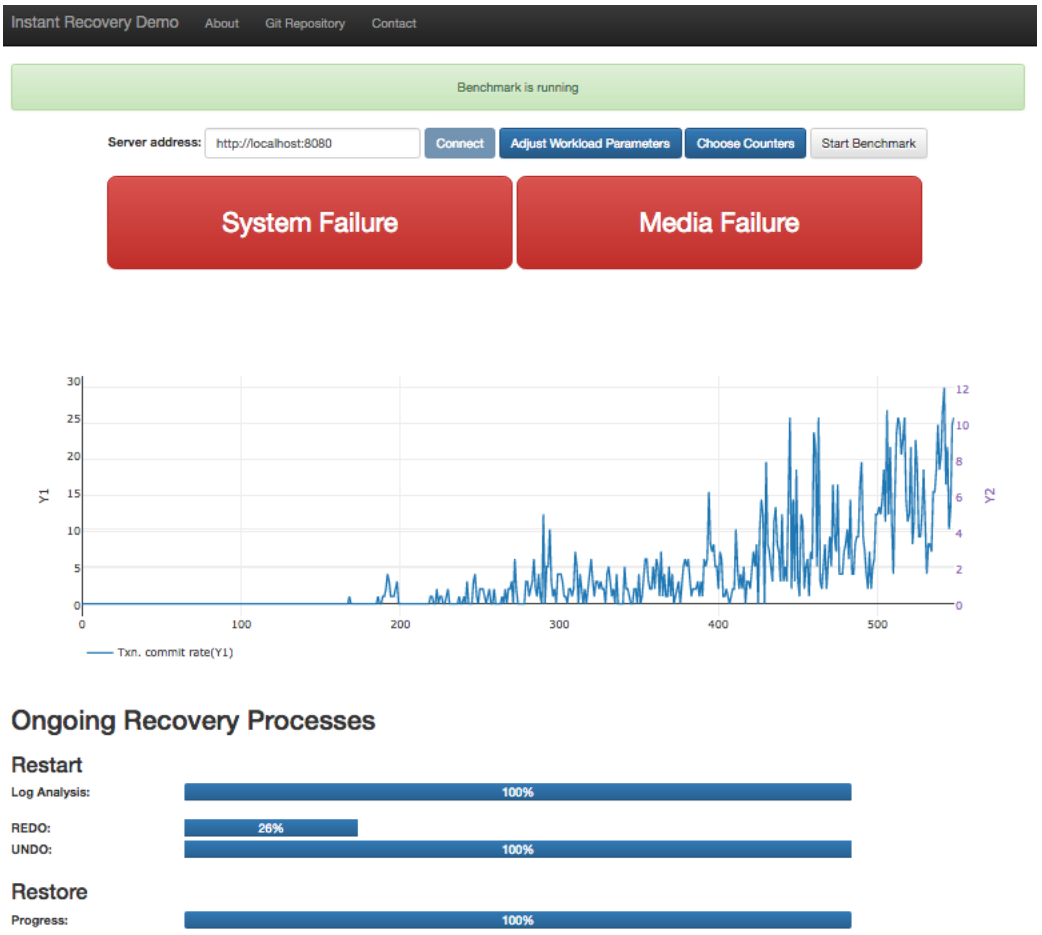

FIGURE 5.2: Instant restart.

small experiment with TPC-C running for only 2 minutes. Such a difference would be even greater in a scenario with the benchmark running for a longer time, which would result in a REDO pass taking longer to process.

5.4 Injecting a Media Failure and Running an Instant Restore Process

In order to introduce a media failure, we started the system and waited until the three steps of instant restart were completed, which can be observed on the three bars of the instant restart section. Such a procedure is not needed, since the system can recover from a media failure while it still recovers from system failures. However, we opted for this procedure in order to observe the system achieve the highest performance.

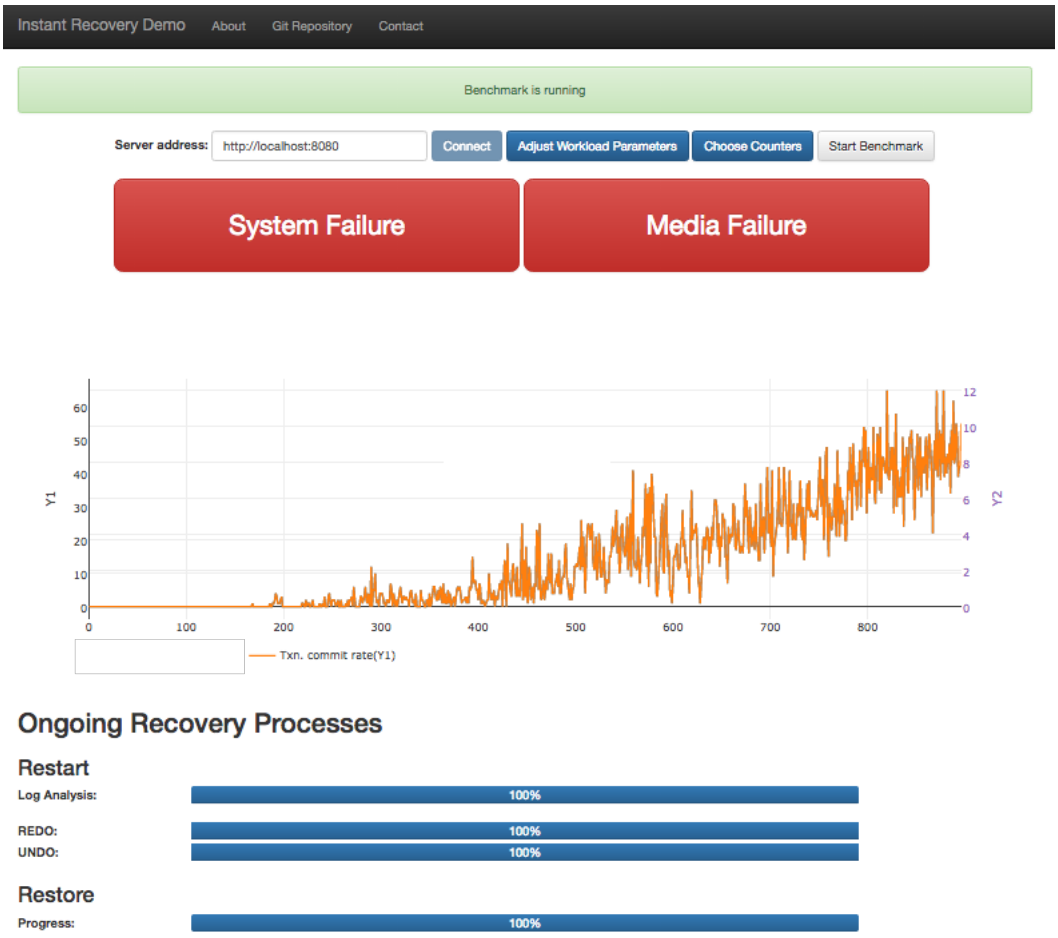

FIGURE 5.3: Instant Restart completed.

Once instant restart was completed, we injected a new media failure by clicking on the `media failure` button. This action starts the instant restore process and its progress can be observed on the bar of the instant restore section.

Figure 5.5 shows the system recovering from a media failure. While instant restore happens, we observe the transaction commit rate. Once the failure occurs, the activity of this operation reduces dramatically. However, the system is still able to commit transactions, which would be impossible by using an offline restore approach. Figure 5.6 shows the state of the same operation after the media is restored, and we can observe how fast the system recovers to commit transactions at a similar rate as before the failure happened.

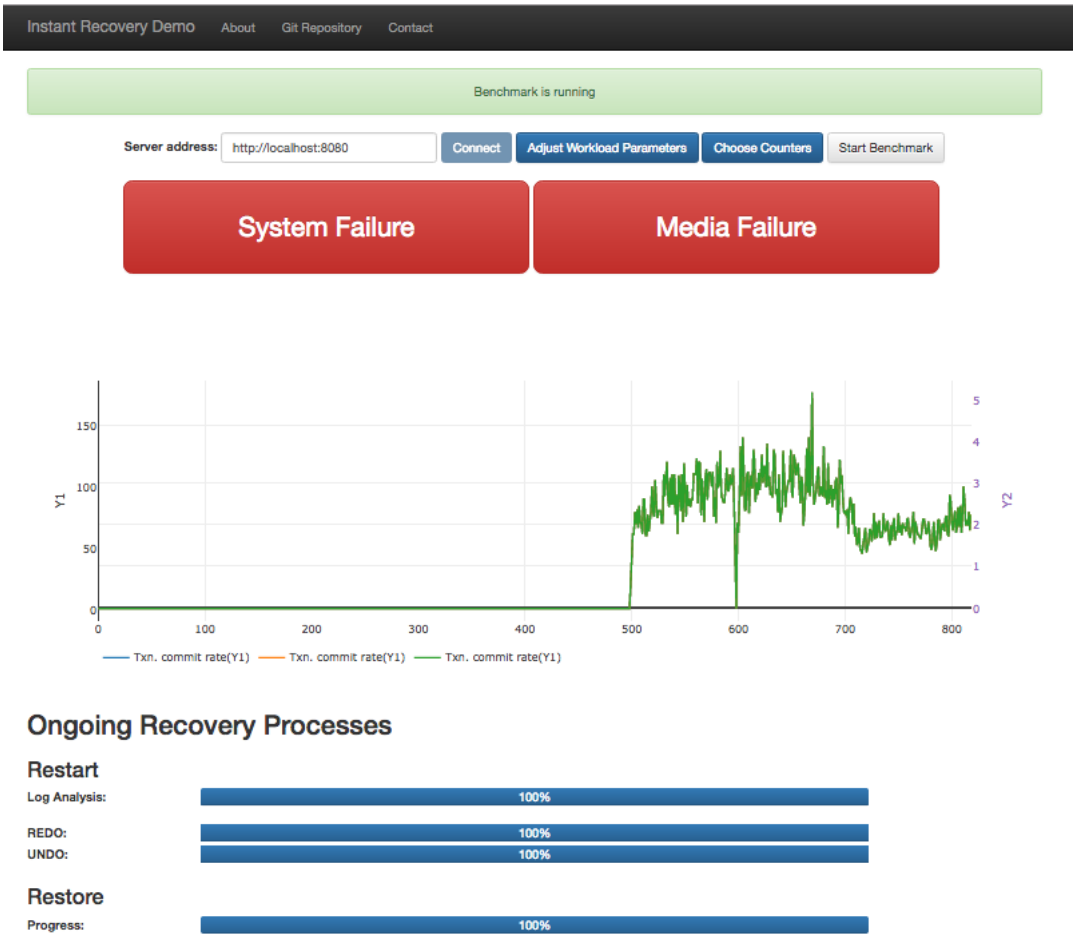

FIGURE 5.4: ARIES Restart completed.

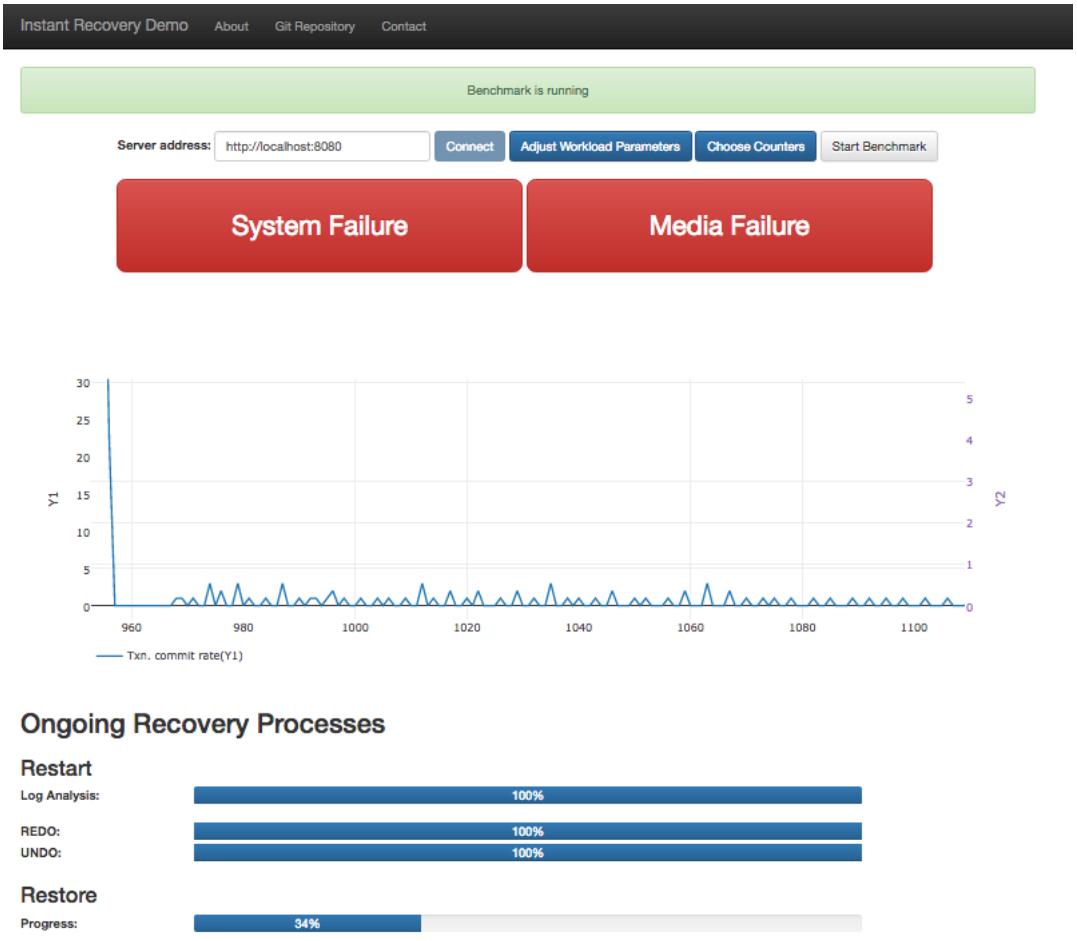

FIGURE 5.5: System state while Instant Restore is in progress.

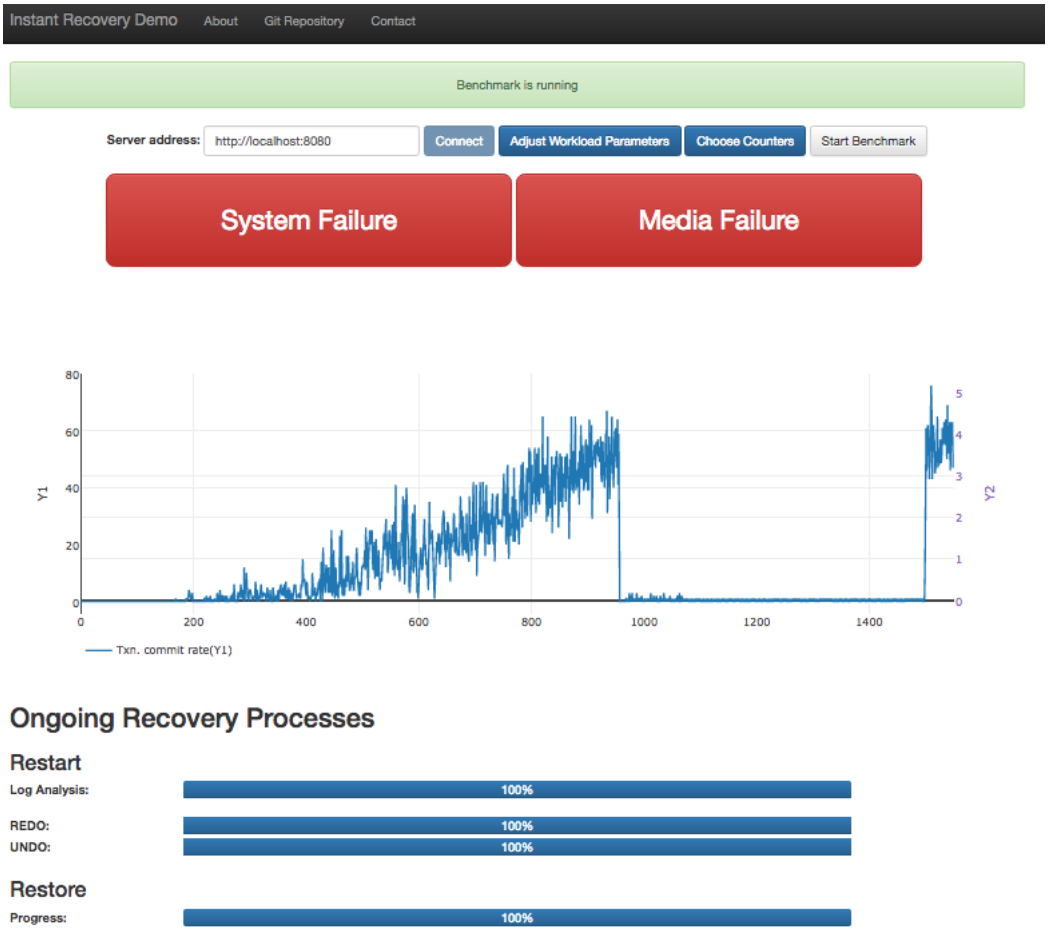

FIGURE 5.6: System state after the system recovered from a media failure.

Chapter 6

Conclusion

As every system eventually fails, recovery techniques have a direct impact on system availability. This thesis implemented a demonstration platform to showcase these database recovery techniques based on write-ahead logging. It focused on modern instant recovery techniques that compete with traditional ARIES. On this platform, the database user is able to run different benchmarks in a prototype database system and observe, through charts and progress bars, how the system recovers from injected failures. The prototype system is able to recover from single-page failures by using Single-Page Recovery, media failures by using Instant Restore, and system failures by using either Instant Restart or ARIES restart.

The main advantage of Instant Recovery is the possibility to recover pages on demand. Because of this possibility, if a crash occurs, the system can be available for new transactions almost instantly after the system restarts. In this scenario, pages on disk requested by new transactions are verified, and if they are considered as failed, they are recovered and then made available for the transactions.

Another advantage of Instant Recovery is the possibility to keep the system available for new transactions even after a media failure. This is possible because Instant Restore can also be executed on demand. In order to be able to recover the media fast enough to execute it on demand, Instant Restore uses a set of strategies to turn the process of restore faster than traditional restore approaches. Among these techniques, there are the indexed full backups and indexed log archive, which are also partially sorted, reducing random reads on the disk, making this technique even faster than an offline traditional restore.

When implementing this demo platform, we chose a two-tier architecture, which allowed us to have our interface separated from our server and be able to run in any Web browser. Furthermore, as our Web server was developed independently of the interface, the development

of further interfaces for different platforms is also independent of the server, since the only requirement for the interface is to be able to execute REST calls and interpret JSON replies.

The server was implemented coupled with the database, which allowed the server to get the values of internal variables of the database and have access to its internal components. This access was important to present the internal state of the DBMS and inject failures in the system in order to start the recovery process. There are three possible failures that can be injected into the DBMS: single-page failure, media failure, and system failure, which simulates the crash of the system. Each of these failures is related to one recovery process, and the hardest to simulate is the recovery of a single page, since we create failures in random pages and these pages will be restored only if the system loads them into the buffer pool. As we cannot predict if the system will use this page again, we cannot guarantee its recovery.

During the execution of our tests with the demonstration platform, we were able to see how fast instant recovery allows new transactions, since it does not wait until the REDO pass is completed. We also compared this behavior to the traditional ARIES restart recovery, which waits for the REDO pass to complete, thus taking longer to accept new transactions. When we observed Instant Restore in action, we were able to see the existence of new transactions on the system even in the presence of a media failure, i.e., while a replacement disk was being restored. Such scenario would not happen if the system used a traditional restore design, since the system would be able to commit new transactions only after the restore process is completed.

Bibliography

[1] 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.

[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.

[3] Gilson Souza. Implementing a Demonstration of Instant Recovery of Database Systems. *TU Kaiserslautern*, 2017.

[4] Zero Storage Manager. URL `https://github.com/caetanosauer/zero`.

[5] 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.

[6] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983. ISSN 0360-0300.

[7] C. Mohan and I. Narang. An efficient and flexible method for archiving a database. *ACM SIGMOD*, 12(1):139–146, 1993.

[8] 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`.

[9] Caetano Sauer, Goetz Graefe, and Theo Härder. Instant restore after a media failure. *CoRR*, abs/1702.08042, 2017.

[10] Goetz Graefe Caetano Sauer and Theo Haerder. Single-pass restore after a media failure. *BTW*, LNI 241, March 2015. URL `http://wwwlgis.informatik.uni-kl.de/cms/fileadmin/publications/2015/SinglePassRestore.pdf`.

[11] TPC. TPC benchmark C standard specification, Feb 2010. URL `http://www.tpc.org/tpcc`.

[12] TPC. TPC benchmark B standard specification, Feb 1994. URL `http://www.tpc.org/tpcb`.

[13] ECMA. The JSON data interchange format., Oct 2013. URL `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[14] Caetano Sauer, Gilson Souza, Goetz Graefe, and Theo Härder. Come and crash our database! – Instant recovery in action. EDBT demo track. *EDBT, 554-557*, 2017.

[15] Boost. Boost.Asio. URL `http://www.boost.org/doc/libs/1_62_0/doc/html/boost_asio.html`.

[16] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-OrientedProgramming*, 1(3):26-49, August 1988. URL `http://arxiv.org/abs/1702.08042`.

[17] Plotly. URL `https://plot.ly/`.

[18] Bootstrap. URL `http://getbootstrap.com/`.