Implementing a Demonstration of Instant Recovery of Database Systems

Gilson Souza dos Santos

Database and Information Systems
University of Kaiserslautern

gilson.s.s@gmail.com

Abstract. We present a Web application that demonstrates Instant Recovery, a family of techniques to enable incremental and on-demand recovery from different classes of failures in transactional database systems. When compared with traditional ARIES recovery, Instant Recovery increases system availability by allowing post-failure transactions to run concurrently with recovery actions, permitting earlier access to data that requires recovery. This mechanism prioritizes data needed most urgently after a failure, thus dramatically reducing the mean time to repair perceived by any individual transaction.

Instant Recovery was implemented in an open-source storage manager and we developed a Web-based system to showcase its recovery capabilities. The system is composed of a Web server, which was developed coupled with the database, and a Web interface, which allows users to execute various benchmarks and observe relevant statistics from the database's internal behavior. These statistics are generated internally by the database and collected by the Web server, which makes them available in a REST API used by our interface to generate dashboards. Using this interface, the user has the ability to start predefined benchmarks in the database and visualize how the internal components of the system behave while the chosen benchmark runs.
.

1 Introduction

Database recovery from system and media failures is an important topic in database availability, as, in traditional design, the time a DataBase Management System (DBMS) takes to recover corresponds directly to the time that the system is unavailable. Recently, Instant Recovery, which is a new recovery technique, was developed in order to increase the availability of the system by allowing the start of new transactions earlier than traditional approaches such as ARIES [1]. The goal of this work is to develop tools to visualize the differences between the new and old techniques in an interactive way.

We developed a demonstration platform that collects information from the DBMS and makes this information visible through a simple Web interface. This platform aims to demonstrate the behavior of the DBMS components – such as transaction, buffer, log, and lock managers – during recovery. This demo was developed using the Zero research prototype, which is a fork of Shore-MT [2] and is available as open source.

As the motivation of the platform is to show the recovery process, we briefly describe the concepts of database recovery. We present ARIES, due to its large use in recent database systems, and Instant Recovery, which is the recovery technique used in Zero. We also present the development of our demo platform, which was built in two separate parts: the server, which is coupled with the database system, and the web interface that controls when the database starts and which information is displayed.

The server was developed to run the benchmarks TPC-B [3] and TPC-C [4] on the Zero storage manager. These standard benchmarks execute a series of operations that are monitored by our application on the server side, making the collected information operations available through a REST API. The Web interface communicates with the server through this API and presents this information graphically, which leads to an online visualization of the state of the system components during the benchmark execution.

In the next section, database recovery is introduced in order to give the background of the metrics and behaviors that are collected at the server side of the demo application. Section 3 presents the demo platform, its architecture, how it was developed, and its functionalities. Finally, section 4 offers some conclusions from this study.

2 Database recovery

Database recovery is the process that guarantees that committed data will remain accessible and consistent in case of a media failure or system crash. In other words, this process ensures the atomicity and durability properties of transactions (the "A" and "D" of ACID) [13]. 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.

There are a variety of system designs for a recovery component, and the ARIES mechanism [1] is the standard for disk-based architectures. ARIES supports fine-granularity locking, it 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 allow new transactions [6]. Instant recovery introduces some modifications to the ARIES design in order to reduce the time to accept new transactions and, therefore, increase system availability.

2.1 ARIES Recovery Method

ARIES recovery relies on a log created during normal processing, i.e., it uses write-ahead logging [1] 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 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 transactions and it is considered as *dirty*. The buffer pool manager registers the dirty pages in the *dirty pages table*, which is used by the recovery system to determine pages in need of recovery. The entries in this table have two fields: PageID, that is 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.

In order to avoid reading the entire log to restore the dirty pages table after a crash, the system periodically saves this table in a log record called *checkpoint*. This log record also contains the state of all active transactions in the system. 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.

2.1.1 Restart after a system failure

When the database restarts after a system failure, the system recovery process starts. At this moment, the information contained in the log is used in three steps: log analysis, REDO and UNDO passes. After the conclusion of the second step, the system is already available for new transactions. The conclusion of the third step concludes the recovery process and brings the system to its last consistent state [1]. The following paragraphs briefly describe each of these phases.

Log Analysis. The routine receives as input the LSN of the master record, which contains the pointer to the last checkpoint taken before the failure. The system recovers the dirty pages table and the state of all active transactions from this checkpoint. After that, the system generates a transaction table based on the transaction status recovered from the checkpoint. This table is later used during the UNDO pass to rollback the transactions active when the failure occurred. The entry fields in the

transaction table are transaction ID, the state of the transaction, the last LSN of the latest log record written by the transaction, and the LSN of the next record to be processed during the rollback.

After log analysis recovers the dirty page table and the transaction table from a checkpoint, it scans log records coming after the last checkpoint and updates these tables 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.

REDO pass. The input to this routine is the RedoLSN, which is the minimum RecLSN from the dirty page table produced during log analysis. No log records are written by this routine. It scans all log records since the RedoLSN point, verifying in each log if the referenced page appears in the dirty page table. If it does, it applies the REDO action described in the log record if and only if its LSN is greater than the PageLSN value.

When REDO ends, the database is reestablished as of the time of system failure, except for updates performed by loser transactions, which are also applied to the database. The UNDO pass solves this problem by rolling back the uncommitted transactions.

UNDO pass. This routine receives as input the transaction table produced during log analysis. After that, the system recovers from each transaction the last generated log record and uses it as a starting point to trace all changes performed by the transaction.

Every log record contains a field called *PrevLSN*. This field points to the previous log record written by the same transaction. Based on this information, the system can identify the log record written by the transaction before its last log record and, by following this sequence, it can trace all log records written by this transaction.

Based on the chain of log records written by the transaction, the system can identify all modifications, which this transaction has done. As these modifications should be undone, the system reacquires all the locks acquired by this transaction in order to block new transactions from acquiring the same locks. This needs to be done because, during the UNDO pass, the system is already available for new transactions [6].

After reacquiring the locks, the UNDO pass starts to undo the changes based on the chain created. When the changes from the first log record 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 these log record's LSN. Therefore, even though the content of the page is rolled back to its previous value, the PageLSN will be different, since it will refer to the compensation log record.

Each CLR contains a pointer called UndoNxtLSN, which points to the log record prior the one it reverses. This pointer is used to track the chain of changes undone by every CLR and based on it is possible to determine precisely how much of the transaction has been undone so far.

An example of the rollback of a page can be seen in Figure 1. The page was restored with the log records from two transactions that changed the values of *Record A* and *Record B* from this page. The execution of these two transactions changed the page LSN to 3, which is the last log record applied by the committed transaction T2. As T1 was not completed when the system crashed, the changes from this transaction are rolled back. When transaction T1 is rolled back, the page LSN is changed to 4, which is the log record of the CLR created by UNDO.

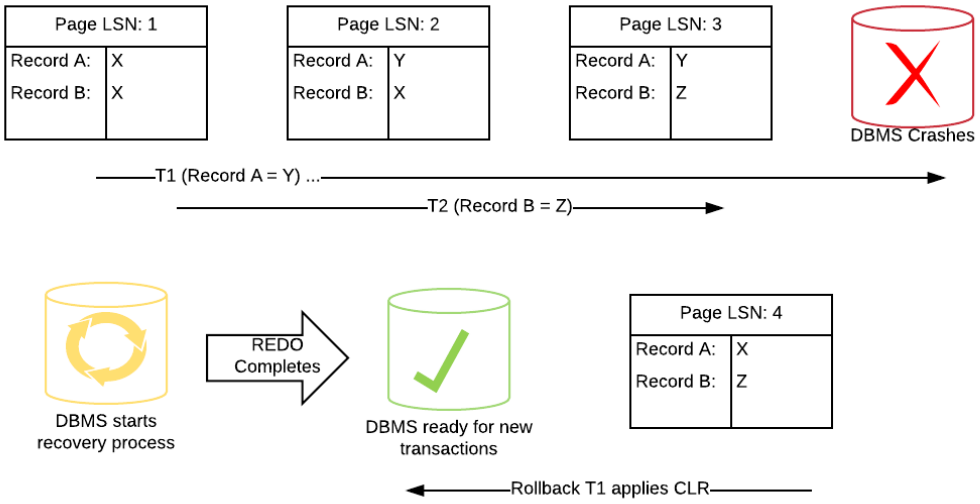

Fig. 1. Rollback of records using CLR

UNDO can run in parallel, running one UNDO thread per transaction. These threads do not interfere with each other in terms of database locks due to the concurrency control, but the access to in-memory data must be controlled using latches. Another way to active the same result is to write the CLRs first, without applying UNDOs to the pages, and then redoing the CLRs in parallel. This approach can also be used for a single transaction [1].

2.2 Instant Recovery

Depending on the workload and the system environment, the recovery technique 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 presents an example of typical times spent in each step of recovery using ARIES. We can see the huge impact in availability if the system enables the execution of new transactions 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.

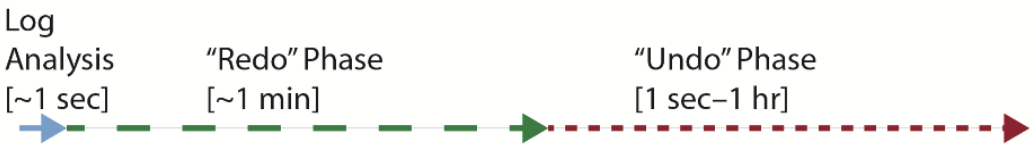

Fig. 2. Example of a time interval spent on recovery [6]

In order to allow new transactions right after the log analysis, Instant Restart defines some changes that must be made in the three passes of system recovery. We briefly explain these changes below.

2.2.1 Log Analysis in Instant Restart

The first step of instant recovery is also log analysis. 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. At the end of log analysis, all pages requiring REDO are registered in the buffer pool, which protects all REDO activity. Similarly, locks protect all UNDO activity [6].

2.2.2 On-demand single-page REDO

After log analysis is finished, the system is available for new transactions. If one of these new transactions requires access to a page registered in Buffer Pool to be recovered, single-page REDO is called to recover this page. After the page is redone, its reference is removed from the list of pages to be redone, avoiding future REDO for this page.

To recover the page, REDO requires the current page in the database and its chain of log records that should be applied to the page. If a log record describing an initial formatting of a newly allocated page is found in the chain of log records, the current page in the database system is ignored. Finally, REDO applies this chain of log records to the required page and, when this process is finished, the page is released to the transaction.

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 memory.

Some page registered in buffer pool may not be required by any post-failure transaction, and the buffer pool may decide to write this page 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 [6].

2.2.3 On-demand Undo

After log analysis, all locks from transactions active at the time of failure are reacquired and their per-transactions log chains ensure their successful rollback. When a lock conflict with a post-failure transaction occurs, on-demand UNDO is invoked and the lost transaction is rolled back. If UNDO touches a page still in need of REDO recovery, the page is recovered as described above. After that, UNDO resumes the transaction rollback. This process continues until the end of 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 described in ARIES, with compensation log records and UndoNext pointers.

3 Demonstration Platform

In order to demonstrate the Instant Recovery technique, we implemented a tool that graphically displays the state of the system as benchmarks execute and recovery is performed. We can observe, for instance, the exact moment when the system starts to accept new transactions and how the number of transactions in the system grows. We also collect other useful information, for the case we want to analyze the behavior of the system operating in real-time, such as the transaction commit and abort rate, the number of pages written by the buffer pool manager, the moment when the REDO and UNDO passes are completed, among many other operations.

This information is available through a Web interface, which connects through a Rest API with a Web server implemented inside the server process. This Web 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 [7]. We can visualize the described operations on Figure 3, which shows the architecture of our demo program.

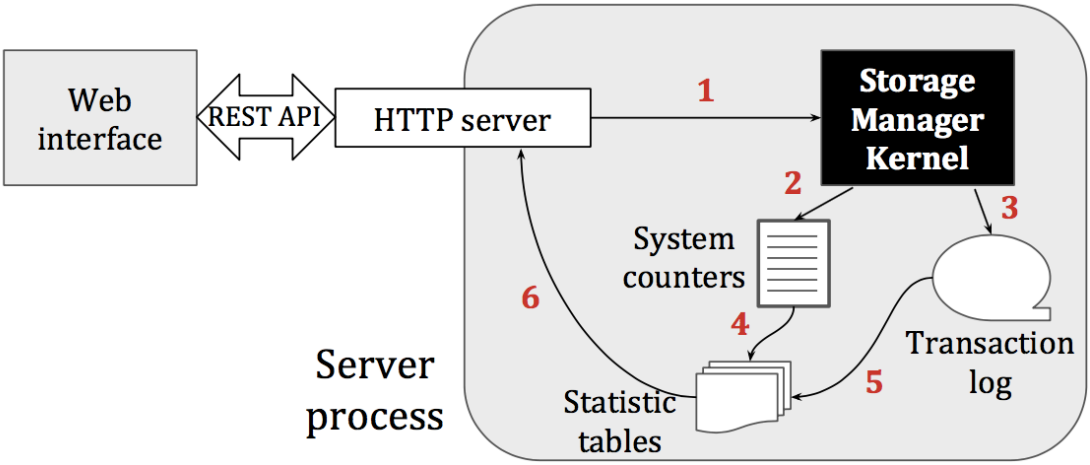

Fig. 3. Architecture of the Demo Program [12]

In order to understand the internal architecture of the Web interface, the HTTP server, and their communication, we describe, as follows, the implementation of these two parts of our demo.

3.1 Server Side Implementation

The main goal of the Web server is to make externally available the internal information from the DBMS while it is running. Besides that, we want to have the option to control the database behavior, such as call internal functions and read internal variables. In order to achieve such goals, we opted to implement our Web server coupled with the database, giving us the possibility to have control over it. The design of Zero helped us to achieve it easier than using a traditional DBMS implementation since Zero contains the transaction storage manager and it also includes the implementation of the benchmarks TPC-B and TPC-C as C++ libraries, besides a set of tools to analyze its log and its behavior.

All the information presented by our tool was already being collected by the database either by counters variables in memory or by the log records written in the log file. In order to make this information available in our REST API, we collected this information from those different sources and standardized it to have the same structure.

3.1.1 Event counters

One of the ways that the database collects information about the operations being executed is through event counters. These counters are variables, which are incremented when a certain operation occurs. To illustrate how counters are produced and collected, we take an example scenario using the buffer manager component.

To exemplify the use of event counters, consider events generated by the buffer manager. Inside the buffer manager component, there is a process called cleaner, which reads all pages from memory and writes the dirty pages to the disk. If two or more pages in the memory can be written in sequence to the disk, the cleaner process can write these pages in only one access to the disk, otherwise, it needs to perform two or more accesses, what increases the cost to write these pages. 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 case.

In our studies, the most interesting counters for our demo are the ones concerned with transaction manager activities, since they show when the system becomes ready to accept new transactions and how the number of transactions 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.

3.1.2 Reading log records

The other approach used to collect information from the DBMS is by analyzing the log file that it produces. As already explained, log records stored in the log file contain information such as checkpoints, page changes, compensations, among some others, and we can read these 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 transactions 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 DBMS 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 incrementing the next column, which results in a table representing the log and grouping logged operations executed every second. Figure 4 represents the creation of this table, as described.

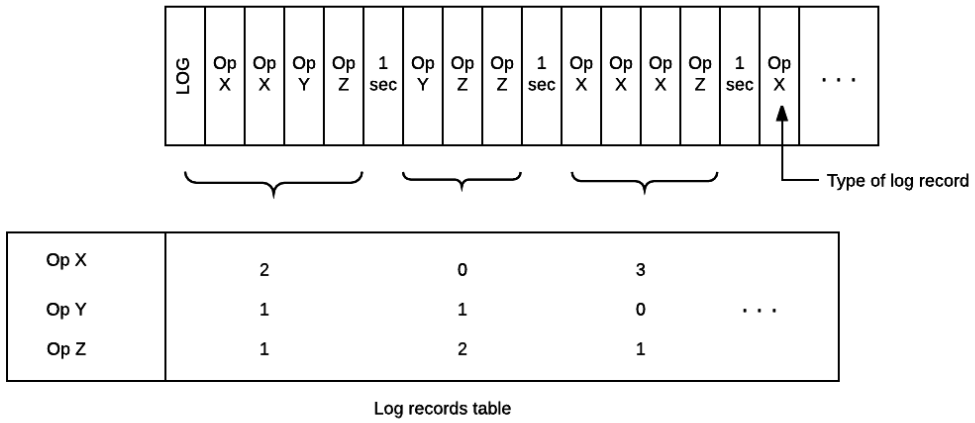

Fig. 4. Creating a table to represent the logged operations.

3.1.3 Counter result table

As we previously described, the system has two sources of information to present the state of the database system: a table based on the log and a list of counters updated in memory. We decided to join these two sources in one table, so clients of the Web server do not have to make more than one request when they want to discover the internal state of the system.

The two sources of the database status are represented in different models: one as a table with a timeline showing the amount 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 existent 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 in a table, which has the same format of the table of logged operations. After every second, a new column 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 JSON format [7], 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.

3.1.4 REST API

Our platform makes the JSON response available through a Web server, which also accepts a singular command to initiate the benchmarks. This Web server was built using *boost.asio* [8], and as already mentioned, it is built in Zero in order to run the benchmarks and collect the database state.

We used Boost.asio to develop a REST API, which is used as the means of communication between the interface and the Web server. There are five routines presented in our API, and they are presented in Table 1, together with their descriptions and parameters.

Name	Description	Parameters
start_kits	Routine responsible to start the database and to run one of the two benchmarks available in Zero: TPC-B and TPC-C.	This routine receives as parameters the name of the benchmark and the duration that the benchmark must run.
is_kits_running	This routine returns the status of the database: If the database is running or not.	No parameters.
counters	Returns all counters in the system (the ones present in memory and the ones created based on the log).	No parameters.
get_stats	It returns only the counters present in memory.	No parameters.
agg_log	This routine returns only the counters based on the log.	No parameters.

TABLE 1. Routines presented in the Webserver API

3.1.5 Handlers

Every routine from the API is tied with one or more internal methods from the storage manager. Some methods are executed periodically and store their results in memory. Thus, if a routine is called, the server only returns the result already present in memory. Other methods are executed on demand, which leads to a method call in the storage manager upon every request.

In order to handle these executions of internal methods, we use classes called *handlers*. These classes are also responsible for avoiding forbidden requests (as to avoid the start of a new benchmark while another benchmark is already running) and for executing the DBMS methods that request a defined order. Such approach also allowed us to centralize the use of database and analysis tools in specific classes, and also isolates their logic.

3.2 Interface Implementation

We used the Model-View-Controller (MVC) [9] pattern to design our Web interface. Besides that, we use JavaScript to access the REST API from the web server, the Plotly library [10] to draw charts and Bootstrap [11] to create the page design and some of its functions.

3.2.1 Model-View-Controller

MVC is a pattern used to isolate business logic from the user interface [9]. Figure 5 presents the iteraction between the three components of this pattern.

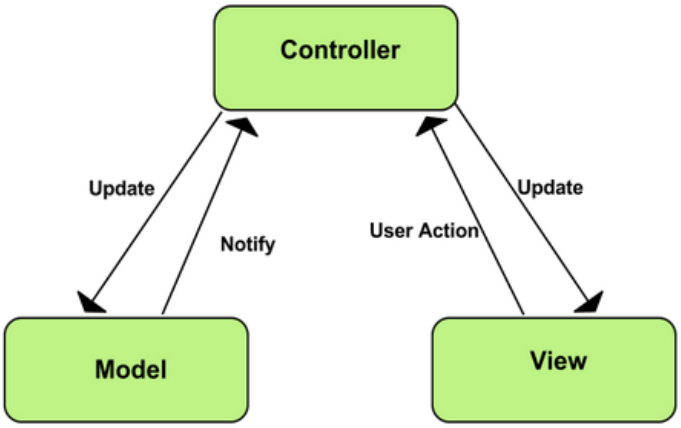

Fig. 5. Model-View-Controller Pattern.

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 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, 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 pipes it in the models existent in the system, updating the dashboards. This process keeps on going until the benchmark finishes.

3.2.2 The interface design 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, which is a framework to create charts, and the design is based on Bootstrap, which is a HTML, CSS, and JavaScript framework to create Web pages. This framework adapts the interface to different devices, such as desktops, tablets, and mobile phones.

Figure 4 shows a screenshot of our demo interface. At the top of the interface, we can see the status of the benchmark, if it is running or not. Below it, we have the options to inform the web server IP address, the workload to be used, the duration of the execution, and the counters to visualize (which can be modified while the benchmark runs).

The screenshot presented in Figure 6 was taken when the database was running the TPC-C benchmark. In this graph, the counters currently being plotted are buffer cleaner I/O time (cleaner_time_io), transaction log flush (xct_log_flush), and the rate of transaction commit (commit_xct_cnt). The latter two are plotted against the right Y-axis.

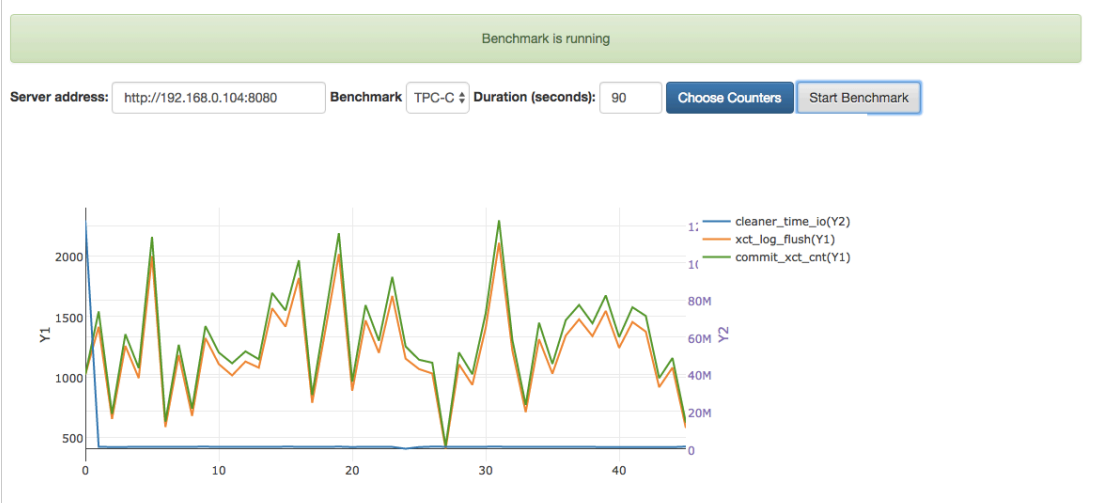

Fig. 6. Demo interface.

There are 276 counters in Zero. The option *Choose Counters* is used to select the counters which will be presented in the chart. This option opens a pop-up menu, which contains a checkbox for each counter. Figure 7 shows this pop-up menu and some of the existent counters.

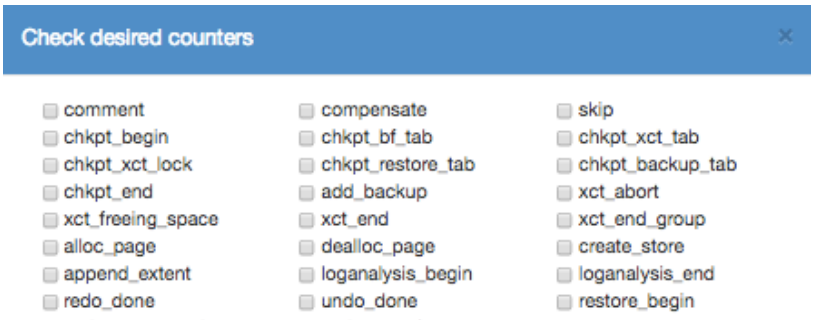

Fig. 7. Counter checkboxes.

As the user may desire to observe two or more counters whose scales differ significantly, we present these counters in two different Y-axes and associate each counter to one of these axes. The axis related to the counter is informed next to the name of the counter, as we can see in Figure 6.

4 Conclusion

In this work, we presented the implementation of the demonstration of Instant Recovery. Instant Recovery is a family of techniques to enable incremental and on-demand recovery from different classes of failures in database systems. Compared with traditional techniques such as ARIES, it brings advantages as allowing transactions to run in the system concurrently to recovery actions, not only permitting earlier access to data that requires recovery but also using the post failures access pattern to actually guide the recovery process.

Instant Recovery was implemented in the Zero storage manager and we developed a Web-based system to showcase its recovery capabilities. The system is composed of a web server, which was developed coupled with the database, and a Web interface, which allows users to execute various benchmarks and observe relevant statistics from the system's internal behavior. These statistics are generated internally in the database by counting events generated by various components and by analyzing the log generated by the DBMS.

The Web server collects the internal database statistics in a table that is sent to the Web client using the JSON format through a REST API. The Web client uses this REST API to control the DBMS and to generate live dashboards from the statistics table.

Users of the demo application are presented with a simple interface, which enables them to start benchmarks in our database with Instant Recovery implemented and visualize live how the internal components of the system behave while the benchmark runs.

5 References

1. Mohan C, Haderle D, Lindsay B, Pirahesh H, Schwarz P, ARIES: a transaction recovery method supporting fine-granularity locking and partial rollback using write-ahead logging, ACM TODS 17(1), 94 162, 1992
2. Johnson R, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. EDBT, 24–35, 2009.
3. TPC. TPC benchmark B standard specification, revision 2.0, 1994. Available at http://www.tpc.org/tpcb, accessed on January/2017.
4. TPC. TPC benchmark C standard specification, revision 5.11, 2010. Available at http://www.tpc.org/tpcc, accessed on January/2017.
5. Sauer C, Härder T, A novel recovery mechanism enabling fine-granularity locking and fast, REDO-only recovery, arXiv:1409.3682, September 2014
6. Graefe G, Guy W, Sauer C, Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, and Media Restore, Morgan & Claypool Publishers, November 2014
7. The JSON data interchange format, First Edition, 2013. Available at: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, accessed on January 2017.
8. Boost Asio, available at http://www.boost.org/doc/libs/1_62_0/doc/html/boost_asio.html, accessed on January 2017.
9. 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/September 1988.
10. Plotly, available at https://plot.ly/, accessed on January 2017.
11. Boostsrap, available at http://getbootstrap.com/, accessed on January 2017.
12. Sauer C, Souza G, Graefe G, Härder T. Come and crash our database! -- Instant recovery in action, EDBT, 554-557, 2017
13. Härder T, Reuter A, Principles of transaction-oriented database recovery, ACM Computing Surveys, 15(4):287-317, December 1983