Indexed Log File: Towards Main Memory Database Instant Recovery

Arlino Magalhaes
Federal University of Ceara
Federal University of Piaui
Fortaleza, Brazil
arlino@ufpi.edu.br

Jose Maria Monteiro
Federal University of Ceara
Fortaleza, Brazil
monteiro@dc.ufc.br

Angelo Brayner
Federal University of Ceara
Fortaleza, Brazil
brayner@dc.ufc.br

Gustavo Moraes
Federal University of Ceara
Fortaleza, Brazil
gustavomoraes94@gmail.com

ABSTRACT

Main Memory Database Systems (MMDBSs) may significantly increment IOPS (Input/Output Operations per Second) rates by avoiding access to secondary memory. This occurs because they maintain the database in Random Access Memory (RAM). Similar to traditional Disk-Based Database Systems (DBSs), MMDBSs are expected to trigger recovery activities after system failures to restore the database to its last consistent state before the failure. Nonetheless, MMDBSs executes the recovery process in an offline way, thus the database becomes available for new transactions only after the full recovery process has been performed. Systems can keep database replicas for high availability. However, replication is not immune to some failure sources that can cause multiple and shared malfunctions. Therefore, software techniques are required to prevent failures and repair crashed systems as soon as possible. This work proposes a novel MMDBS instant recovery process which makes MMDBS able to schedule new transactions simultaneously with the recovery activities. In order to validate this new approach, simulations with a prototype implemented on Redis have been conducted over Memtier benchmark. The achieved results evidence the suitability of the proposed recovery mechanism.

1 INTRODUCTION

Main Memory Databases (MMDBs), or In-Memory Databases (IMDBs), can provide very high throughput rates given that the primary data are located in memory. In that manner, MMDB reduces secondary memory I/O bottleneck, and can consequently speed up data access. Moreover, the development of new memory technologies has provided a larger storage capacity with lower costs. The fact that the database resides in volatile storage influences the design approaches adopted by MMDBs, such as query processing, concurrency control, recovery after crashes, data storage, and indexing. For this reason, these systems are designed to optimize access to main memory instead of secondary memory, as with traditional disk-resident systems [7, 21, 24].

MMDBs provide very high IOPS given that the primary database is handled in volatile storage. However, the database residing in a volatile memory makes these systems much more sensitive to system failures than conventional disk-resident database systems. The recovery mechanism is responsible for restoring the database to the most recent consistent state before a system failure has occurred. In this way, after a system crash, the recovery manager loads the last valid checkpoint (a prior database backup copy) and then starts to execute all actions recorded in the log file forward from the checkpoint record [10, 12, 13].

Accordingly, the recovery process for most MMDBs is performed offline, meaning that the database and its applications only become available for new transactions after the full recovery process is completed. One may claim that systems can keep database replicas for high availability. In fact, with the advent of high-availability infrastructure, recovery speed has become secondary in importance to runtime performance for most MMDBs [7, 12, 22]. Nevertheless, replication is not immune to human errors and unpredictable defects in software and firmware that are a source of failures and can cause multiple and shared problems [18, 21].

In this sense, this paper proposes an instant recovery approach for OLTP MMDBs. Said approach allows MMDBs to schedule new transactions immediately after the failure during the recovery process, giving the impression that the system was instantly restored. The main idea of instant recovery is to organize the log file in a way that enables efficient on-demand and incremental recovery of individual database tuples.

It is important to note that the existing MMDBs recovery strategy has two deficiencies that make instant recovery impossible. First, the recovery process uses a sequential log file. The recovery in the sequential log is not incremental and requires full recovery before any tuple can be accessed. This scenario does not allow the system to execute an on-demand transaction during recovery, which means that new transactions can only start executing after the recovery process has finished. The second problem is the random access pattern in the sequential log for restoring tuples individually. The sequential log has efficient record writes, but it has inefficient reads for individual log records. A full log scan must be done to restore a given tuple individually [12, 18].

The instant recovery technique presented in this work builds the log file as an index structure. This log organization enables an efficient restoration of a tuple. A single fetch on the indexed log can restore one tuple. Thus, the system can use the indexed log to recover a database by restoring tuple by tuple incrementally. This technique naturally supports database availability because a new transaction can access a tuple immediately after the tuple is restored, i.e., transactions do not have to wait for a full recovery to access restored tuples. We have empirically evaluated the proposed instant recovery approach in order to show its efficiency and suitability to be implemented in MMDBs. The workload used for the experiments belongs to Memetier benchmark.

The remainder of this paper is organized as follows. Session 2 provides an overview of MMDB recovery. Section 3 discusses related work. Section 4 presents the proposed approach for database instant recovery. Section 5 discusses the results of empirical experiments. Finally, Section 6 concludes this paper.

2 BACKGROUND

Most main memory database systems implement logical logging technique which records higher-level database operations, such as inserting a record in a table. MMDBSs do not record Before images of modified tuples, i.e., they produce only Redo log records to reduce the amount of data written to secondary storage. The commit processing uses group commit, i.e., it tries to group multiple log records into one large I/O. SSD is the log device of choice for almost all systems in order to increase the I/O performance [12, 22, 26].

The recovery component of most MMDBs asynchronously produces a consistent checkpoint, commonly called snapshot. Snapshot is equivalent to a materialized database state in an instant of time by means of a Copy-on-Update method (COU, for short) [4, 6]. At the checkpoint beginning, MMDBS enters into a COU mode. Thereafter, the recovery component starts to scan all tuples in database tables at system run-time. Three bits are used to identify if a row was inserted, deleted, or updated after the checkpoint generation has begun. Inserted tuples are disregarded. Before an update or delete, the original content of a tuple is copied to a shadow table. The shadow version is removed after it is scanned. A background process serializes the snapshot to secondary memory until the checkpoint ends [2, 12].

Whenever a system crash occurs in an MMDB, the primary copy of the database is lost. In this case, MMDBs recover the database by loading the last valid checkpoint. Thereafter, the recovery component starts to execute the actions recorded in the log file forward from the checkpoint record. The recovery component activities are briefly illustrated in Figure 1. All actions of committed transactions are flushed to the log file on secondary memory by the Logger component. Periodically, the Checkpoint component produces a snapshot on secondary storage. After a failure, the Restorer component loads the snapshot into memory and then replay the log file. After the recovery process has finished, the database is available for new transactions [7, 22].

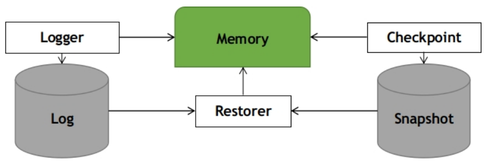

Figure 1: MMDBS recovery component architecture.

3 RELATED WORK

Hekaton [5], VoltDB [20], HyPer [9], SAP HANA [8] and SiloR [26] are examples of modern MMDBSs that perform the recovery activities discussed in Section 2. Nevertheless, those systems do not execute new transactions until the full recovery is completed.

PACMAN [22] and Adaptive Logging [23] utilize a dependency graph between transactions performed to identify opportunities for database recovery in parallel. After a failure, they use a dependency graph to generate a scheduled execution to replay the log records. The schedule allows transactions to be executed in parallel, following the constraints of the dependency graph. Those systems must wait for the full database recovery to service new transactions.

The Log-Structured Merge tree (LSM-tree) [15] provides low-cost indexing for a file that has a high rate of record insertions and deletions. However, the LSM-tree access method uses a buffer to avoid multiple I/Os in secondary memory for frequently referenced pages. This approach is not suitable for writing log records since they require immediate and atomic persistence during commit processing.

FineLine [17–19] presents an instant database restoration technique. This technique uses a partitioned index in the log to write records efficiently. The partition index may search for multiple partitions to retrieve a page. After a crash, the recovery process loads pages incrementally from a backup device. While a set of pages is loaded, the records in the log file that are related to that set are probed. This approach also can recover pages on-demand for transactions. New transactions can perform as soon as necessary pages are restored.

4 A NOVEL INSTANT RECOVERY MECHANISM

In this section, firstly the architecture of the proposed instant recovery mechanism is described. Thereafter, the proposed log data structure is discussed. Finally, the recovery algorithm based on the proposed indexed log structure is detailed.

4.1 The Architecture

Figure 2 proposes an architecture to implement our MMDB instant recovery approach, which uses an indexed log structure. This section provides an overview of the main components that comprise the architecture and their interactions.

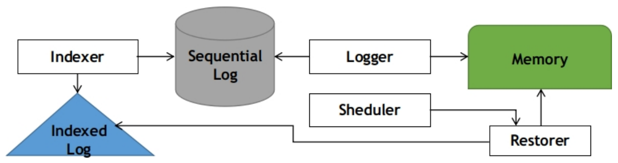

Figure 2: Architecture for in-memory database instant recovery using an indexed log.

Below, we present a brief description of the main components of the architecture discussed in the following subsections.

- **Logger**: writes transaction update actions in a log file on secondary memory. The records are flushed sequentially in an append-only file whose writes must be synchronized to transaction commit.
- **Indexer**: indexes records from sequential log to indexed log. The records are indexed in a B-tree asynchronously to transaction commit.
- **Restorer**: restores tuples from a failed database by replaying records from indexed log. Tuples can be restored incrementally and on-demand.
- **Scheduler**: during the recovery process, requests the Restorer component for tuples (that have not yet been restored into memory) requested by new transactions.

4.2 The Logging Strategy

The proposed approach for MMDB instant recovery uses two logs: a sequential log (Figure 3 (a)), and an indexed log (Figure 3 (b)). Each record in the sequential log represents an update performed on a tuple by a transaction. During transaction processing, transaction update records are appended to the sequential log file by the Logger component. Each transaction generates Redo records that are kept in a thread-local. During the commitment, all log records generated by a transaction are appended atomically on the sequential log. This scheme ensures log consistency to recover the database.

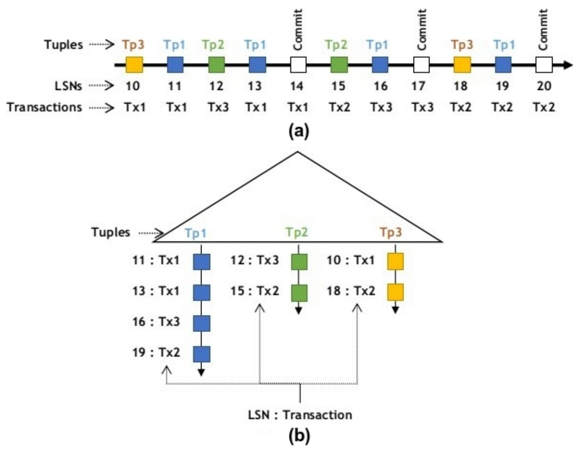

Figure 3: Sequential log (a), and indexed log (b).

The proposed recovery scheme requires efficient log reading to fetch the records to redo a given tuple during recovery. For this reason, the logging strategy implements an indexed log. The index structure is a B-tree in which each node contains a tuple ID and the update records generated by transaction updates in the tuple. Only one probe on B-tree can retrieve all the necessary records to restore a single tuple. The Indexer component is responsible for indexing records from the sequential log to the indexed log. The indexing is asynchronous to transaction commit, i.e., a transaction does not need to wait for the log indexing to confirm its writing in the data. Records can be removed from the sequential log after they are indexed in the B-tree. However, the sequential log is maintained to ensure consistent database recovery in the event of index corruption. In this case, the system must build a new indexed log from the sequential log. This process will delay the start of recovery.

The primary purpose of instant recovery is to restore the database efficiently, without degrading the transaction throughout provided by the system. The indexed log requires random writes, while a sequential log has a sequential write pattern. Writing records to a sequential log file is potentially faster than doing so to an indexed log file. For this reason, in our approach, log records are written to the sequential file and, periodically, flushed to the indexed log file. The indexing process occurs asynchronously to the transaction commit operation so as not to degrade the transaction processing. It is important to highlight that restoring an individual tuple by indexed log requires only one fetch on B-Tree, while restoring a single tuple by sequential log requires a full scan of the log file. Therefore, our recovery technique only uses the indexed log to recover the database after a failure.

FineLine [17–19] uses an instant recovery technique that allows efficient write of log records by a partitioned index. However, probes on log require inspecting multiple partitions to restore a page. This approach can delay recovery. The number of partitions can be reduced by intermediate merges. However, this process can interfere with the transaction processing performance. Our log organization is simpler and writes/reads records efficiently. Transactions must wait only for writes on the sequential log to commit. The log indexing does not interfere with transaction processing since records are indexed asynchronously to transaction commit. During recovery, only one fetch on B-tree can restore a tuple individually.

As it was exemplified in Figure 3, transactions Tx1, Tx2, and Tx3 generated log records for updates performed in tuples Tp1, Tp2, and Tp3. Sequential log (Figure 3 (a)) stores the records flushed by the three transactions. The log records with LSN 11, 13, 16, and 19 represent the last update performed in tuple Tp1, for example. A fetch on the indexed log can retrieve the records of LSN 11, 13, 16, and 19 to redo the tuple Tp1. The absence of an index implies the necessity of a full scan on the sequential log to restore Tp1.

4.3 The Recovery Algorithm

After a system failure, the system should initiate the database recovery by restoring tuples through the indexed log. However, the record indexing process is asynchronous to the transaction commit. As a result, some records on the sequential log may not have been indexed before a failure. Therefore, immediately before starting recovery, the system must verify if any records have not yet been indexed. Indexer component must index those records to ensure the recovery consistency. When this process ends, recovery can begin and new transactions can be performed. Thus, the Restorer component begins redoing tuples by traversing the indexed log B-Tree. Each visit to a B-Tree node can retrieve the update records to redo a tuple. After visiting all B-Tree nodes, all database tuples are restored, and the recovery process is completed.

The indexed log recovery scheme can naturally support availability since new transactions can be executed immediately after restoring their required tuples. Furthermore, this recovery scheme can service new transactions whose necessary tuples have not yet been loaded into memory during recovery. When a transaction requires tuples, the system checks if the tuples are stored in memory. If they are not in memory, the Scheduler component must request the Restorer for these tuples on-demand. Then, the recovery manager should pause the incremental recovery (the traversing in B-tree) and begin fetching the necessary tuples for the transaction from the indexed log. After the transaction's tuples are restored, they are marked as restored, the transaction can run, and the system can continue the incremental recovery.

4.4 The Evaluation Prototype

The instant recovery approach proposed in this paper was implemented in Redis 5.0.7 [16] to evaluate the feasibility of indexing for log replay. The evaluation prototype can be downloaded[1]. Redis is an open-source in-memory data structure store used as an in-memory key-value database. Redis is written in ANSI C.

Persistence in Redis can be achieved through snapshotting and logging. In snapshotting, the database is asynchronously transferred from memory to secondary storage at regular intervals as a binary dump using the Redis RDB Dump File Format.

[1]https://drive.google.com/drive/folders/1LTbtY36O0kWIpxZBM-hc1BPvIjICuy2F

In logging, a record of each operation that modifies the database is added to an append-only file (AOF). Redis can automatically rewrite the AOF in the background when it gets too big [16]. Our prototype uses only the AOF, i.e., the RDB was disabled. Moreover, the system does not rewrite the log.

During transaction performing, each update operation generates a log record that contains basically its command, key, and value. For example, the operation *SET(K1, V1)* stores the value *V1* with the key *K1* and generates the log record fields *SET*, *K1*, and *V1*. Each record is written in the AOF atomically only at a committed time and at the same order each in which the command was performed. Our prototype uses the AOF from Redis, i.e., we did not need to implement a sequential log.

The records must be copied from the sequential log to the indexed log periodically. The indexed log is a B-tree implemented in Berkeley DB 4.8 [14]. Berkeley Database (Berkeley DB or BDB) is a software library intended to provide a high-performance embedded database for key/value data.

5 EVALUATION

We have empirically evaluated the instant recovery approach proposed in this research. We used the Memtier Benchmark to perform the tests in Redis. All experiments shown in this paper were executed with 4 worker threads on Intel Core i7-9700k CPU 3.60GHz x 8. The system has 64GB of RAM and 400GB of SSD Kingston SA400S37 as a persistent storage device. The operating system was Ubuntu Linux 18.04.2 LTS.

5.1 Mentier Benchmark

Memtier is a high-throughput benchmarking tool for Redis developed by Redis Labs. This tool has a command-line interface that provides a set of customization and reporting features to generate various workload patterns. It can launch multiple worker threads, with each thread driving a configurable number of clients. The tool can control the ratio between read and write operations. Moreover, it offers control over the pattern of keys used by the operations (e.g., random and sequential patterns). Memtier provides options to set the number of total requests per client or the number of seconds to run a test. The tool offers other options for configuring custom workloads [1, 11]. Memtier has already been used in several scientific works, such as in [25] and [3].

5.2 Recovery Experiments

The first group of experiments was focused on measuring the time to fully recover a database, availability to process transactions after a system failure, time to run a workload entirely, and logging overhead. These experiments were performed on a database containing 99, 507 kyes that generated an 11.8GB sequential log file containing 160 million records. Additionally, an indexed log was generated along with this sequential log using the recovery technique proposed in this work. For each experiment, the system was shut down to simulate a failure. At the database restart, as soon as the recovery process was been triggered, a workload would be submitted. Thus, one could measure transaction throughput and recovery time from system restart.

The key goal was to compare the proposed instant recovery approach to the traditional main memory database recovery. However, we also tested our instant recovery scheme in different scenarios to confirm the following expectations about our technique: (1) an indexed log must be employed to incrementally and on-demand recover the database, and (2) asynchronous indexing of log records must be used to avoid transaction processing overhead. Thus, the experiments have been conducted in the three following scenarios: *(i)* Sequential Log Recovery - SLR; *(ii)* Asynchronous Indexed Log Instant Recovery - AILIR; *(iii)* Synchronous Indexed Log Instant Recovery - SILIR.

The SLR scenario (traditional recovery) uses only a sequential log. In this scenario, transaction update records are written to a sequential log file during transaction processing. The recovery process recovers the database by scanning the entire log file. Transactions can be performed only after the recovery is complete. The AILIR scenario (our approach) uses a sequential log + indexed log. In AILIR, transaction update records are written in a sequential log during transaction processing and stored asynchronously to transaction commit in an indexed log. The SILIR scenario (scenario derived from AILIR) uses only an indexed log. In SILIR, transaction update records are written directly to an indexed log synchronously to the transaction commit. After a failure, for both scenarios ii (AILIR) and iii (SILIR), the recovery manager must traverse the B-tree to recover the database, and transactions can perform during recovery. The SILIR scenario was created to measure the log indexing overhead during transaction processing and instant recovery processing.

For each scenario mentioned above, three experiments were performed by different types of workload: *(i)* read-only workload that contains only read operations, *(ii)* read-write workload that has read and write operations in the 5:5 ratio, and *(iii)* write-only workload that has only write operations. These three workloads were simulated using Memtier benchmark that used 4 worker threads, with each thread driving 50 clients. Each client made 170,000 requests in a random pattern.

Figure 4 shows the results of recovery experiments for the three scenarios (SLR, AILIR, and SILIR) performing the read-only workload, denoted Scenarios Read-Only. The vertical dashed lines in the figure indicate the final recovery time of the respective color approach. AILIR and SILIR recovered the database at the same time interval (85 seconds) since both techniques use the same algorithm to recover the database. They recovered before SLR which took 91 seconds to recover. In addition, they were available for new transactions since the database restart and before SLR which can execute transactions only after full recovery. Those two approaches had a quite similar throughput during the workload performing. Moreover, after the recovery, the three scenarios had a similar throughput. This was because there were no records to flush to the log file. AILIR and SILIR had a slightly lower throughput during recovery due to access to the indexed log. AILIR and SILIR performed the entire workload before SLR, as they can process new transactions during recovery.

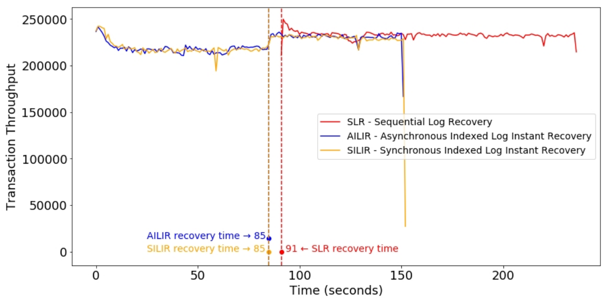

Figure 4: Recovery experiments - Scenarios Read-Only.

The results of recovery experiments for the three scenarios mentioned in this paper performing the read-write workload (denoted Scenarios Read-Write) are in Figure 5. The vertical dashed lines in the figure indicate the final recovery time of the respective color approach. The AILIR approach did not overload the throughput of transactions since its throughput was similar to that of the default approach (SLR). This result was already expected because AILIR and SLR flush log records to secondary memory in a similar manner, except that AILIR additionally indexes the log records. However, the indexing did not interfere with the transaction throughput because it is performed asynchronously to transaction commit. SILIR had the worst performance due to its synchronous log indexing, i.e., a transaction must wait for indexing to confirm its writes. Although SLR recovered the database before AILIR, AILIR was the fastest approach to finish the workload execution. This result was achieved because AILIR has asynchronous indexing and can process transactions while the system is recovering. In addition, the client application did not notice the AILIR recovery, giving the impression that the recovery was instantaneous.

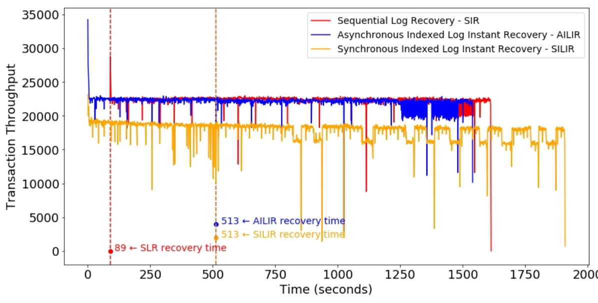

Figure 5: Recovery experiments - Scenarios Read-Write.

Figure 6 presents the results of recovery experiments for the three scenarios performing the write-only workload (denoted Scenarios Write-Only). These results are similar to those in Figure 5. Except for the fact that SILIR recovered the database faster than AILIR. However, this fact did not influence AILIR's performance. The AILIR approach had better performance since it was the fastest approach to finish the workload execution without overloading the transaction throughput. It had a very similar throughput to default recovery.

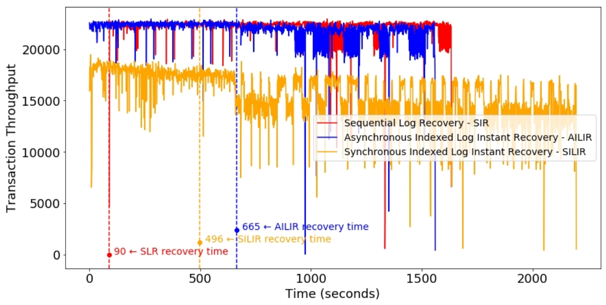

Figure 6: Recovery experiments - Scenarios Write-Only.

5.3 Scalability Experiments

We ran further experiments in which the proposed recovery strategy deals with different log file sizes. The goal is to observe the behavior and performance of the recovery strategy when the log file increases in size. These experiments were performed on the following four databases:

(1) DB1: containing 49, 866 kyes that generated a 5.9GB sequential log file containing 80 million records.
(2) DB2: containing 99, 507 kyes that generated an 11.8GB sequential log file containing 160 million records.
(3) DB3: containing 198, 067 kyes that generated a 23.6GB sequential log file containing 320 million records.
(4) DB4: containing 392, 041 kyes that generated a 47.3GB sequential log file containing 640 million records.

These four databases mentioned above have a ratio of approximately 1:1600 between keys and log records. Each sequential log was generated along with an indexed log using the recovery technique proposed in this work. The experiments used the same scenarios handled in the previous section (Section 5.2): *(i)* Sequential Log Recovery - SLR; *(ii)* Asynchronous Indexed Log Instant Recovery - AILIR; *(iii)* Synchronous Indexed Log Instant Recovery - SILIR. An experiment was performed for each of the four databases (DB1, DB2, DB3, and DB4) in each scenario (SLR, AILIR, and SILIR). For each experiment, the system was shut down to simulate a failure. At the database restart, as soon as the recovery process was been triggered, a workload would be submitted. The workload was simulated using Memtier benchmark that used 4 worker threads, with each thread driving 50 clients. Each client made 300,000 requests. The workload had 5:5 read and write operations in a random pattern. From the system restart, we measured the average transaction throughput during the execution of the workload, the recovery time, and the total workload execution time.

Figure 7 presents the recovery time obtained in each test. As expected, these results show that the number of database keys and log records can delay the recovery process in all approaches, as the recovery time increases with the size of the database. The instant recovery approaches (AILIR and SILIR) had a similar recovery time because they use the same recovery technique. However, AILIR was slightly faster than SILIR in all tests. This is because the weight of synchronous indexing by SILIR influences the overall performance of the system. SLR recovered the database faster than AILIR and SILIR.

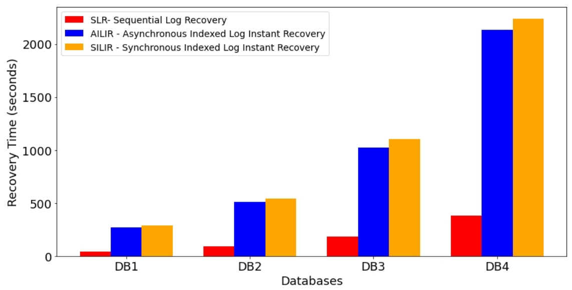

Figure 7: Scalability experiments - Recovery time.

Figure 8 shows that AILIR interferes very little in the throughput of transactions. This is evidenced by the fact that the AILIR average throughput, for the execution of a given workload, remained similar to that of the standard approach (SLR) in all experiments. On the other hand, SILIR's average transaction throughput was much lower than that of AILIR, proving that asynchronous indexing is essential for a better performance of the instant recovery technique. The throughput difference between AILIR and SLR may occur because the Indexer component blocks the sequential log to read it. In the meantime, transactions must wait for the lock to be released before writing records to the log. Nevertheless, the approach proposed in this paper has the advantage of high availability. It can perform new transactions since the system restart, while the standard approach must wait for full database recovery to perform new transactions.

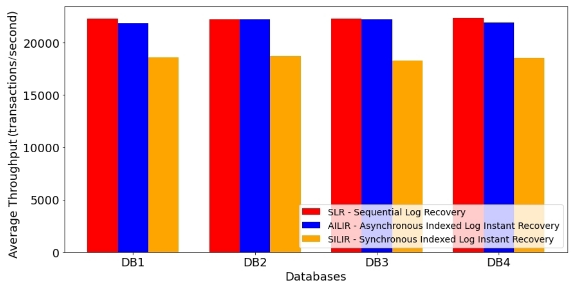

Figure 8: Scalability experiments - Average throughput.

The results of the experiments in Figure 9 show that the execution time of AILIR's workload is slightly longer than that of SLR. Although the experiments in Figures 4, 5, and 6 have shown that AILIR performed the workload faster than SLR, the experiments in Figure 9 show that this behavior changes with higher workloads. The workload of the experiments in Figure 9 (300,000 operations) is 1.7x greater than the workload of the experiments in Figures 4, 5, and 6 (170,000 operations). This is because SLR's transaction throughput is slightly higher than that of AILIR, as shown in Figure 8.

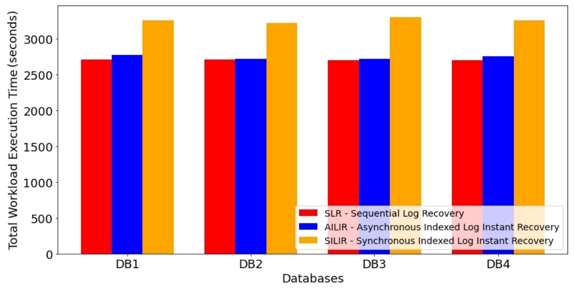

Figure 9: Scalability experiments - Total workload execution time.

6 CONCLUSION

This paper proposed an instant recovery approach for main memory database systems. The proposed approach allows new transactions to run concurrently to the recovery process. Our approach implements an indexed log to fetch tuples directly on the log to restore data incrementally. Consequently, new transactions are scheduled as soon as required tuples are restored into the main memory database. Furthermore, the proposed recovery mechanism restores data on-demand since it restores tuples for new transactions whose data has not yet been restored.

The results show that instant recovery reduces the perceived time to repair the database since transactions can be performed since the system is restarted. In other words, it can effectively deliver tuples that new transactions need during the recovery process. The experiments also analyzed the impact of using a log indexed structure on transaction throughput rates in an OLTP workload benchmark.

REFERENCES

[1] Memtier Benchmark. 2020. GitHub - RedisLabs/memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool. Retrieved August 26, 2020 from https://github.com/RedisLabs/memtier_benchmark

[2] Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. 2011. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. Association for Computing Machinery (ACM), 265–276.

[3] Wenqi Cao, Semih Sahin, Ling Liu, and Xianqiang Bao. 2016. Evaluation and analysis of in-memory key-value systems. In *2016 IEEE International Congress on Big Data (BigData Congress)*. IEEE, 26–33.

[4] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. 1984. *Implementation techniques for main memory database systems*. Vol. 14. Association for Computing Machinery.

[5] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1243–1254.

[6] Margaret H Eich. 1986. Main memory database recovery. In *Proceedings of 1986 ACM Fall joint computer conference*. IEEE Computer Society Press, 1226–1232.

[7] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin Levandoski, Thomas Neumann, Andrew Pavlo, et al. 2017. Main Memory Database Systems. *Foundations and Trends® in Databases* 8, 1-2 (2017), 1–130.

[8] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. *ACM Sigmod Record* 40, 4 (2012), 45–51.

[9] Florian Funke, Alfons Kemper, Tobias Mühlbauer, Thomas Neumann, and Viktor Leis. 2014. HyPer Beyond Software: Exploiting Modern Hardware for Main-Memory Database Systems. *Datenbank-Spektrum* 14, 3 (2014), 173–181.

[10] Le Gruenwald, Jing Huang, Margaret H Dunham, Jun-Lin Lin, and Ashley Chaffin Peltier. 1996. Recovery in main memory databases. (1996).

[11] Redis Labs. 2020. Redis Labs | The Best Redis Experience. Retrieved October 06, 2020 from https://redislabs.com

[12] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory oltp recovery. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE, 604–615.

[13] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[14] Michael A Olson, Keith Bostic, and Margo I Seltzer. 1999. Berkeley DB.. In *USENIX Annual Technical Conference, FREENIX Track*. 183–191.

[15] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[16] Redis. 2020. Redis. Retrieved August 26, 2020 from https://redis.io

[17] Caetano Sauer. 2017. *Modern Techniques for Transaction-oriented Database Recovery*. Ph.D. Dissertation. University of Kaiserslautern, Kaiserslautern, Germany.

[18] Caetano Sauer, Goetz Graefe, and Theo Härder. 2017. Instant restore after a media failure. In *Advances in Databases and Information Systems*. Springer, 311–325.

[19] Caetano Sauer, Goetz Graefe, and Theo Härder. 2018. FineLine: log-structured transactional storage and recovery. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2249–2262.

[20] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[21] Kian-Lee Tan, Qingchao Cai, Beng Chin Ooi, Weng-Fai Wong, Chang Yao, and Hao Zhang. 2015. In-memory databases: Challenges and opportunities from software and hardware perspectives. *ACM SIGMOD Record* 44, 2 (2015), 35–40.

[22] Yingjun Wu, Wentian Guo, Chee-Yong Chan, and Kian-Lee Tan. 2017. Fast Failure Recovery for Main-Memory DBMSs on Multicores. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 267–281.

[23] Chang Yao, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, and Sai Wu. 2016. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1119–1134.

[24] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1920–1948.

[25] Yiying Zhang and Steven Swanson. 2015. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–10.

[26] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*, Vol. 14. 465–477.