Main Memory Databases Instant Recovery

Arlino Henrique Magalhaes de Araujo
Supervised by Jose Maria Monteiro and Angelo Brayner
Federal University of Ceara
Federal University of Piaui
Fortaleza, Brazil
arlino@ufpi.edu.br

ABSTRACT

The recovery process in main memory database systems (MMDBs) run in an offline way. Thus, MMDB only becomes available for new transactions after the complete recovery process has finished. Some MMDBs maintain database replicas for assuring high availability after systems failure. Nonetheless, a database replication mechanism is not immune to failures as well. For that reason, recovery techniques are required to repair failed systems as quickly as possible. This work proposes an instant recovery strategy for MMDBs, which makes MMDBs able to process transactions immediately after the recovery engine is triggered. The proposed approach rebuilds the database incrementally and on-demand. Besides, a novel checkpoint technique is proposed to interfere as little as possible in the system performance. The checkpoint technique can also act during the recovery process so that the next recoveries are faster in the face of successive failures. In order to validate the approach, simulations with a prototype implemented on Redis have been conducted over Memtier benchmark. Preliminary results evidence the suitability of the proposed recovery mechanism.

1 INTRODUCTION

It is a matter of fact that Main Memory Databases provide very high throughput rates since the primary database is handled in main memory. Nevertheless, databases residing in a volatile memory are much more sensitive to system failures than traditional disk-resident. The recovery mechanism is responsible for restoring the database to the most recent consistent state before a system failure [2, 7, 8].

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 might claim that systems are capable of keeping 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. Nevertheless, high availability infrastructure is not immune to human errors and unpredictable defects in software and firmware that are a source of failures and might cause multiple and shared problems. Besides, it require an additional cost to the system infrastructure. [2, 7, 12, 15].

Proceedings of the VLDB 2021 PhD Workshop, August 16th, 2021. Copenhagen, Denmark.

This paper proposes an instant recovery approach for MMDBs. The proposed 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. Furthermore, the paper presents a log record propagation scheme (checkpoint) to accelerate recovery and free up log space. The scheme uses an extension the fuzzy checkpoint approach to try not to degrade system performance. The checkpoint can also propagate records during recovery and thus accelerate next recoveries in case of successive failures.

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

2 BACKGROUND AND RELATED WORK

Most MMDBs implements logical logging technique which records higher-level database operations, such as inserting database tuples. MMDBs produce only Redo log records of modified tuples 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 [15, 18].

The MMDB checkpoint materializes log logical operations to physical data on a checkpoint file. However, most MMDBs produces a consistent checkpoint file equivalent to a materialized database state in an instant of time, commonly called snapshot [1–3].

Whenever a system crash occurs in an MMDB, the primary copy of the database is lost. Thus, the recovery manager should load the last checkpoint into memory and redo log records [2, 14, 15].

Hekaton [1], VoltDB [14], HyPer [4], SAP HANA [3], and SiloR [18] are examples of modern MMDBs that perform the recovery activities mentioned above. Nevertheless, those systems do not execute new transactions until the full recovery is completed.

PACMAN [15] and Adaptive Logging [16] utilize a dependency graph between transactions performed to identify opportunities for database recovery in parallel. Those systems must wait for the full database recovery to service new transactions.

The Log-Structured Merge tree (LSM-tree) [10] provides low-cost indexing for a file that has a high rate of record insertions and deletions. 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.

Sauer et al. [12, 13] present a technique for instant restoring a disk-resident database. This technique uses a partitioned index to write log records efficiently. After a crash, the recovery loads pages from a backup device and their corresponding log records from the indexed log. New transactions are allowed as soon as their necessary pages are restored. As a disadvantage, when the number of partitions increases, the system may search for multiple partitions to retrieve a page, which might delay the recovery.

In the paper [6] the authors provide a survey of techniques for implementing recovery in MMDBs. Besides, the authors describe the main features of recovery mechanisms delivered by well-known MMDBs.

3 THE INSTANT RECOVERY MECHANISM

The existing MMDBs recovery strategies use a sequential log that makes instant recovery impossible. The recovery by a 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 since the required tuples for the transaction can only be accessed after the recovery process has finished [2, 7, 12].

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 since 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. Figure 1 shows the architecture to implement the proposed MMDB instant recovery approach. The next subsections discuss the main components of the architecture and their interactions.

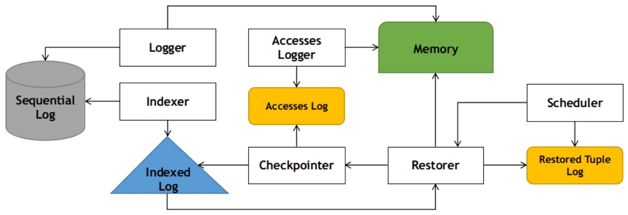

Figure 1: Architecture of the instant recovery mechanism.

3.1 The Logging Strategy

The proposed approach for MMDB instant recovery employs two log files: a sequential log (Figure 2 (a)), and an indexed log (Figure 2 (b)). Each record in the sequential log represents an update performed on a tuple by a transaction. During transaction processing, 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 by the Logger component. The Log Sequence Number (LSN) represents the order in which a record was stored. This scheme ensures log consistency to recover the database.

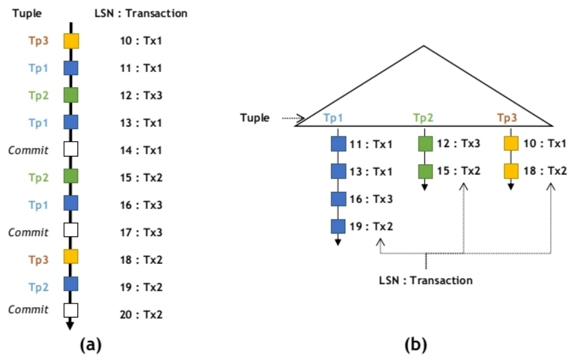

Figure 2: Sequential log (a), and indexed log (b).

The proposed recovery scheme requires efficient log reading to fetch the records in order to redo a given tuple during recovery. For this reason, the logging strategy implements an indexed log. The index structure is an extension to B^+-tree in which each node contains a tuple ID and the log records generated by transaction updates on that tuple. Therefore, the log granularity is tuple. One probe on B^+-tree retrieves all records to restore a single tuple. The Indexer component indexes records from the sequential log to the indexed log asynchronous to transaction commit, i.e., a transaction does not need to wait for the log indexing to commit its execution. 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 delays the start of recovery process.

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 into a sequential log file is potentially faster than doing so into an indexed log file. For this reason, in our approach, log records are written to the sequential file since they have efficient record writes. Periodically, the log records are flushed from sequential log to indexed log file. The indexing process occurs asynchronously to the transaction commit operation so as not to degrade the processing. Therefore, the proposed log organization can flush log records efficiently to a sequential log during transaction processing and it needs only one fetch on the indexed log to retrieve all log records required to restore one tuple.

Figure 2 illustrates logging process. Observe that transactions Tx1, Tx2, and Tx3 generated log records for updates performed in tuples Tp1, Tp2, and Tp3. Sequential log 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.

3.2 The Checkpoint Strategy

During transaction processing, the Accesses Logger component stores tuple access information. The Checkpointer component uses those information to identify the most frequently accessed tuples. These tuples are associated with most of the generated log records. Periodically, the Checkpointer starts the checkpoint process that is an extension of the fuzzy checkpoint approach. The checkpoint propagates the record actions in the indexed log of the most frequently used tuples.

For each tuple T, in which T is one of the most frequently accessed tuples, the Checkpointer retrieves the set S of log records contained in node N of B^+-tree (in the indexed log) associated with T. The set S is able to redo the tuple T entirely in case of failure. Then, a new log record L, whose effect is equivalent to that of the set S to redo tuple T, is generated. Finally, log record L replaces set S on node N. This process is useful to decrease the number of log records to be processed during recovery. As a result, the checkpoint can accelerate the recovery process and liberate log space. The checkpoint is performed asynchronously to the transaction processing.

3.3 The Recovery Process

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. The Indexer component must index those records to ensure 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. When a tuple is redone, its key is marked as restored. 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 Scheduler component checks whether the tuples have already been restored. If they are not in the memory, the Scheduler 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.

During recovery, the checkpoint propagates log record actions similarly to the algorithm shown in Section 3.2. However, the propagation is applied to the log records of each restored tuple, rather than just the most frequently used tuples. Therefore, after a tuple is restored, the Restorer requires the Checkpointer to perform a checkpoint for the log records for that tuple. The contents of the restored tuple are used to generate a new log record that replaces the log records in the B^+-tree node used to redo the tuple. If successive failures occur, the next recovery will process fewer log records.

3.4 The Evaluation Prototype

The instant recovery approach proposed in this paper was implemented in Redis 5.0.7 [11] 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, has a simple architecture, and its source code is easy to understand. These characteristics facilitated the development of the prototype.

Redis uses an append-only file (AOF) to write log records and generates snapshots at regular intervals as a binary dump using the Redis RDB Dump File Format. Redis can automatically rewrite the AOF in the background in case its size exceeds the optimal [11]. The prototype uses the AOF from Redis, i.e., we did not need to implement a sequential log. The RDB was disabled in the prototype. Moreover, the system does not rewrite the log.

The B^+-tree of the indexed log was implemented using Berkeley DB 4.8 [9]. Berkeley Database (Berkeley DB or BDB) is a software library intended to provide a high-performance embedded database for key/value data. BDB allows the specification of the underlying organization of the data in various database implementations (e.g., b-tree, hash, queue, and recno). The checkpoint component proposed in this work is still under development. Thus, the experiments presented in the next session do not include this module.

4 RECOVERY MECHANISM EVALUATION

We have empirically evaluated the proposed instant recovery mechanism in order to present its efficiency and suitability to be implemented in MMDBs. All experiments 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. We used the Memtier benchmark [5, 17] to perform the tests. Memtier is a high-throughput benchmarking tool for Redis developed by Redis Labs. The tool offers options to generate various workload patterns. Memtier has already been used in several scientific works, such as this one [17], for example.

4.1 Preliminary Experiments

The experiments were focused on measuring the time to fully recover a database, the availability to process transactions after a system failure, the time needed to run a workload entirely, and logging overhead. These experiments were performed on a database containing 99, 507 keys 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 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 MMDB 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

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

database, and (2) the asynchronous indexing of log records must be adopted 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. 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 the transaction commit in an indexed log. The SILIR scenario (which is 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. The SILIR scenario was created to measure the log indexing overhead during transaction processing and instant recovery processing. For each scenario, the experiments were performed on workloads with a 5: 5 ratio between read and write operations. They were simulated using the Memtier benchmark which operated 4 worker threads, with each thread driving 50 clients. Each client made 170,000 requests in a random pattern.

The results of recovery experiments are in Figure 3. The vertical dashed lines in the figure indicate the final recovery time of the respective color approach. Besides, it 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 the SLR recovered the database before AILIR, the AILIR was able to perform transactions since system restart and 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. Additionally, the client application did not notice the AILIR recovery, giving the impression that the recovery was instantaneous.

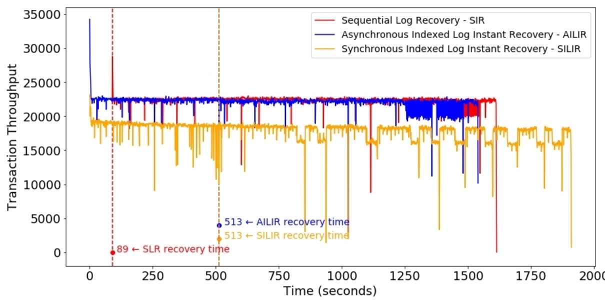

Figure 3: Experiment results

5 CONCLUSION

This paper proposed an instant recovery approach for MMDBs. The approach allows new transactions to run concurrently to the recovery process. The approach takes benefit of using a log file with a B^+-tree structure. Thus the recovery mechanism is able to seek tuples directly on the log file to rebuild the database in an on-demand and incremental fashion. New transactions are scheduled as soon as required tuples are restored into the MMDB.

The results show that instant recovery reduces the perceived time to repair the database, seeing that 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. We believe that adding a checkpoint module to the prototype developed in this work will increase system availability and provide faster database recovery.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] Arlino Magalhaes, Jose Maria Monteiro, and Angelo Brayner. 2021. Main Memory Database Recovery: A Survey. *ACM Computing Surveys (CSUR)* 54, 2 (2021), 1–36.

[7] 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.

[8] 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.

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

[10] 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.

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.