FEDERAL UNIVERSITY OF CEARA

NATURE SCIENCE CENTER

DEPARTMENT OF COMPUTING

COMPUTER SCIENCE POSTGRADUATE PROGRAM

ARLINO HENRIQUE MAGALHÃES DE ARAÚJO

MAIN MEMORY DATABASE INSTANT RECOVERY

FORTALEZA

2022

ARLINO HENRIQUE MAGALHÃES DE ARAÚJO

MAIN MEMORY DATABASE INSTANT RECOVERY

Thesis presented in the Computer Science Postgraduate Program of the Nature Science Center at the Federal University of Ceara, as a partial requirement for obtaining the title of doctor in Computer Science. Concentration area: Databases

Advisor: Prof. Dr. José Maria da Silva Monteiro Filho

Co-advisor : Prof. Dr. Angelo Roncalli de Alencar Brayner

FORTALEZA

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A687m Araújo, Arlino Henrique Magalhães de.
Main Memory Database Instant Recovery / Arlino Henrique Magalhães de Araújo. – 2022.
135 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2022.
Orientação: Prof. Dr. José Maria da Silva Monteiro Filho.
Coorientação: Prof. Dr. Angelo Roncalli de Alencar Brayner.

1. Database. 2. In-memory database. 3. Main memory database. 4. System failure. 5. Instant recovery. I. Título.

CDD 005

ARLINO HENRIQUE MAGALHÃES DE ARAÚJO

MAIN MEMORY DATABASE INSTANT RECOVERY

Thesis presented in the Computer Science Postgraduate Program of the Nature Science Center at the Federal University of Ceara, as a partial requirement for obtaining the title of doctor in Computer Science. Concentration area: Databases.

Approved in: 06/06/2022.

EXAMINATION BOARD

_____

Prof. Dr. José Maria da Silva Monteiro Filho (Advisor)
Federal University of Ceara (UFC)

_____

Prof. Dr. Angelo Roncalli de Alencar Brayner (Co-advisor)
Federal University of Ceara (UFC)

_____

Prof. Dr. Javam de Castro Machado
Federal University of Ceara (UFC)

_____

Prof. Dr. Eduardo Cunha de Almeida
Federal University of Parana (UFPR)

_____

Prof. Dr. José de Aguiar Moraes Filho
Federal Data Processing Service (SERPRO)

To my parents, sisters, wife and sons who did not measure the effort so that I could achieve this phase of my life.

ACKNOWLEDGEMENTS

I always thought the acknowledgments section the worst part of thesis writing. This is because it may be difficult to put life in a regression analysis and quantify what happens in our life.

First of all, I thank God for guiding me and giving me peace of mind to face the difficulties and continue pursuing my goals. Today I see that all the obstacles I faced brought me to where I am now.

I thank my parents for raising me and giving me the support I needed to pursue my goals and, above all, for showing me the importance of studying. Furthermore, I thank them and my sisters for the moments of happiness that are very important in the trajectory of a person's life.

I thank my wife and sons for accompanying me during the doctorate. The family's love and moments of happiness help us to face the difficulties and uncertainties we encounter.

I thank my advisor and co-advisor for their patience in guiding me in this learning process that was the doctorate. The various discussions (and reviews) of the articles paved the way for the thesis.

I also have to thank my UFC friends, especially from the ARIDA laboratory, who partnered during my studies and relaxed moments at lunch and afternoon coffee. During all these years, we shared friendship, companionship, difficulties, uncertainties and victories.

Finally, I thank UFPI, the educational institution where I work, which gave me the opportunity to take time off from work to do my doctorate.

“In fact, what is essential in education is not the doctrine taught, it is the awakening.”

(Ernest Renan)

RESUMO

A tecnologia de Bancos de Dados em Memória manipulam o banco de dados primário em memória principal para prover alta vazão de dados e baixa latência. Entretanto, a volatilidade da memória faz os bancos de dados em memória muito mais sensíveis a falhas. O conteúdo do banco de dados é perdido em tais falhas e, como resultado, o sistema deve ficar indisponível por um longo tempo até o processo de recuperação ter sido terminado. Assim, novas técnicas de recuperação são necessárias para recuperar bancos de dados falhados o mais rápido possível. Esta tese apresenta *MM-DIRECT* (*Main Memory Database Instant RECovery with Tuple consistent checkpoint*), uma técnica para recuperação de bancos de dados em memória capaz de escalonar transações simultaneamente ao processo de recuperação do banco de dados desde o reinício do sistema. Dessa maneira, é dada a impressão de que o banco de dados foi recuperado instantaneamente. Além disso, essa abordagem de recuperação implementa um checkpoint consistente a nível de tupla para reduzir o tempo de recuperação. Para validar a abordagem proposta, experimentos foram executados em um protótipo implementado no banco de dados Redis. Os resultados mostram que a técnica de recuperação instantânea efetivamente provê altas taxas de vazão de transações durante o processo de recuperação e durante a processamento normal do banco de dados.

Palavras-chave: bancos de dados; banco de dados em memória; falha de sistema; recuperação instantânea.

ABSTRACT

Main Memory Databases (MMDBs) technology handles the primary database in Random Access Memory (RAM) to provide high throughput and low latency. However, volatile memory makes MMDBs much more sensitive to system failures. The contents of the database are lost in these failures, and, as a result, systems may be unavailable for a long time until database recovery process has been finished. Therefore, novel recovery techniques are needed to repair crashed MMDBs as quickly as possible. This thesis presents MM-DIRECT (Main Memory Database Instant RECovery with Tuple consistent checkpoint), an MMDBs recovery technique able to schedule transactions simultaneously with the database recovery process at system startup. Thus it is giving the impression that the database is instantly restored. The approach implements a tuple-level consistent checkpoint to reduce the recovery time. In order to validate the proposed approach, experiments have been performed in a prototype implemented on the Redis database. The results show that the instant recovery technique effectively provides high transaction throughput rates even during both the recovery process and normal database processing.

Keywords: database; in-memory database; main memory database; system failure; instant recovery.

LIST OF FIGURES

LIST OF TABLES

ALGORITHM LIST

LIST OF ABBREVIATIONS AND ACRONYMS

ACC	Action-Consistent Checkpoint
ACID	Atomicity, Consistency, Isolation, and Durability
ARIES	Algorithms for Recovery and Isolation Exploiting Semantics
ARIES/CSA	ARIES for the Client-Server Architecture
ARIES/IM	ARIES for Index Management
ARIES/KVL	ARIES using Key-Value Locking
ARIES/LHS	ARIES for Linear Hashing with Separators
ARIES/NT	ARIES for Nested Transactions
ARIES-RRH	ARIES with Restricted Repeating of History
ART	Adaptive Radix Tree
BW-chain	Backward chain
BWC-tree	Backward chain tree
CLR	Compensation Log Record
COU	Copy-on-Update
CPU	Central Process Unit
CS	Client-Server
CSB+-Trees	Cache Sensitive B+-Trees
CSS+-Trees	Cache Sensitive Search Trees
DBDS	Disk-Based Database System
DBMS	Database Management System
DSM	Decomposition Storage Model
DTT	Dirty Tuple Table
EAW	Editable Atomic Writes
EOT	End of Transaction
FAST	Fast Architecture Sensitive Tree
FOEDUS	Fast Optimistic Engine for Data Unification Services
FSM	Flexible Storage Model
HD	Hard Disk
HOT	Height Optimized Trie
HTAP	Transaction/Analytical Processing
HTM	Hardware Transactional Memory

I/O	Input/Output
IMDB	In-Memory Database
IOPS	Input/Output Operations per Second
LHS	Linear Hashing with Separators
LSN	Log Sequence Number
MMDB	Main-Memory Database
MM-DIRECT	Main Memory Database Instant RECovery with Tuple consistent checkpoint
MRBTree	Multi-rooted B+-Tree
MVCC	Multi-Version Concurrency Control
NSM	N-ary Storage Model
NTA	Nested Top Action
NUMA	Non-Uniform Memory Access
NVM	Non-Volatile Memory
OCC	Optimistic Concurrency Control
OLAP	Online Transaction Processing
OLC	Optimistic Lock Coupling
OLTP	Online Transaction Processing
pB+-Trees	Prefetching B+-Trees
PCC	Pessimistic Concurrency Control
PCM	Phase-Change Memory
PLP	Physiological Partitioning
POSIX	Portable Operating System Interface
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
SIMD	Single Instruction Multiple Data
SSD	Solid-State Drive
STTMRAM	Spin-Transfer Torque Magnetic RAM
TCC	Transaction-Consistent Checkpoint
TM	Transactional Memory
T-SQL	Transact-SQL
TuCC	Tuple Consistent Checkpoint
TuCC-MFU	Tuple Consistent Checkpoint for MFU tuples
WAL	Write-Ahead Logging

WBL	Write-Behind Logging
ZAB	ZooKeeper Atomic Broadcast

CONTENTS

1 INTRODUCTION

In-Memory Database (IMDB), or Main-Memory Database (MMDB), technology has proved to be an efficient alternative for a real-time view of operational data (OLTP-style) and high-performance critical-mission situations due to high throughput rates and low latency provided by these systems. This is because, in such a system, the database resides in random access memory. Additionally, the development of new memory technologies has provided a larger storage capacity with lower costs. To illustrate our claim, memory storage capacity and bandwidth are growing at a rate of 100% every three years. In the meanwhile, RAM costs are falling by a factor of 10 every five years (TAN *et al.*, 2015; ZHANG *et al.*, 2015). Moreover, other recent hardware/architecture improvements can potentially provide better performance to MMDBs with low overhead, such as NUMA architecture (LEIS *et al.*, 2014), SIMD instructions (WILLHALM *et al.*, 2009), RDMA networking (MITCHELL *et al.*, 2013), hardware transactional memory (LEIS *et al.*, 2014), and non-volatile memory (ARULRAJ; PAVLO, 2017). Such advances have boosted the development of MMDBs. Section 2.4 presents the core technologies that leverage MMDBs.

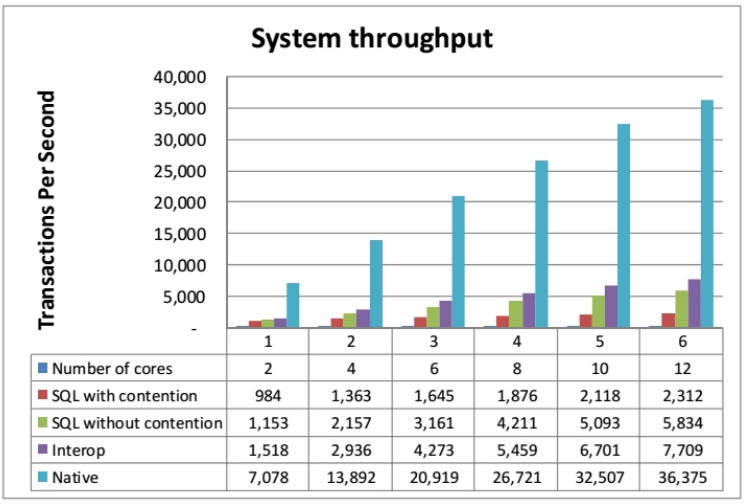
The technological foundations of MMDB arose in the 1980s (DEWITT *et al.*, 1984; HAGMANN, 1986; EICH, 1986; EICH, 1987a; LEHMAN; CAREY, 1987; EICH, 1987b). In this period, some products were developed, such as IMS/Fast Path (STRICKLAND *et al.*, 1982; GAWLICK; KINKADE, 1985), MARS MMDB (EICH, 1987b), System M (SALEM; GARCIA-MOLINA, 1990), TPK (LI; NAUGHTON, 1988), OBE (BITTON *et al.*, 1987), and HALO (GARCIA-MOLINA; SALEM, 1992), just to name a few. Much of this research has focused on improving the performance of traditional disk-based Database Management System (DBMS) or fitting the entire database into the main memory. However, the high price and limited capacity of RAM at that time made the application of MMDB an infeasible solution. Nonetheless, in the nineties, the advances in hardware technology re-generated interests in MMDB research (GARCIA-MOLINA; SALEM, 1992; LEVY; SILBERSCHATZ, 1992; ZHANG *et al.*, 2015; TAN *et al.*, 2015). Some commercial MMDBs emerged and were employed in performance-critical applications, mainly in telecommunication and stock market applications. Examples of such MMDBs are Dali/DataBlitz (JAGADISH *et al.*, 1994; BAULIER *et al.*, 1999), ClustRa (HVASSHOVD *et al.*, 1995), TimesTen (LAHIRI *et al.*, 2013; TEAM, 1999) and P*Time (CHA; SONG, 2004).

A straightforward and simple assumption of how to implement an MMDB is to increase the size of the main memory of a disk-based database system to fit the entire database in

the cache. This is an easy way to minimize access to secondary memory and improve system performance. However, simply replacing the storage layer of a disk-resident database with the main memory does not make it as efficient as an MMDB. This is because the complex components of traditional disk-based databases focus on improving access to secondary storage. For example, the buffer manager must check if every data request is in the cache and which is not. MMDBs are designed to optimize access to main memory instead of secondary memory. They are not dependent on disk logical layout (e.g., block size) and can store data in any format to achieve high throughput rates (HAZENBERG; HEMMINGA, 2011; GUPTA *et al.*, 2014; HARIZOPOULOS *et al.*, 2019). Some MMDBs give a direct pointer to the main memory to access an object. This is called swizzling (STONEBRAKER, 1991; GARCIA-MOLINA; SALEM, 1992). The main architectural attributes of an MMDB must be redesigned considering access to the main memory to achieve its potential, such as data storage, indexing, concurrency control, query processing, durability, and recovery (HAZENBERG; HEMMINGA, 2011; GUPTA *et al.*, 2014).

To illustrate the potential of MMDBs, the work (DIACONU *et al.*, 2013) provides a comparison and evaluation between the traditional SQL Server and Hekaton (an in-memory engine). Both products are from Microsoft. Figure 1 shows the results of the experiments performed in the SQL Server and Hekaton running a typical customer workload. Both databases residing entirely in memory. With 12 cores, Hekaton had a throughput of 36,375 transactions per second, 15.7X performance improvement over the SQL Server, which had only 2,312 transactions per second. This result is due to the traditional design of disk-resident systems (e.g., latch contention).

Figure 1 – Hekaton vs. SQL Server scalability



	1	2	3	4	5	6
Number of cores	2	4	6	8	10	12
SQL with contention	984	1,363	1,645	1,876	2,118	2,312
SQL without contention	1,153	2,157	3,161	4,211	5,093	5,834
Interop	1,518	2,936	4,273	5,459	6,701	7,709
Native	7,078	13,892	20,919	26,721	32,507	36,375

Source: Diaconu *et al.* (2013).

1.1 The research problem

MMDBs provide very high throughput rates since the primary database is handled in volatile storage. However, this feature renders in-memory systems more vulnerable to some sources of disruption that do not occur in disk-resident systems. The fact that the database resides in volatile storage is the most critical problem for MMDBs when a crash occurs. For this reason, recovery after failures is a critical feature in MMDBs. The MMDB recovery component should be effective and efficient. Effectiveness ensures that after a crash, the consistent database state, which existed before the failure, is restored. In turn, efficiency ensures that the recovery process achieves its goal in a short time to make the database available as soon as possible. The recovery process involves other activities performed during database transaction processing, e.g., logging and checkpoint (HÄRDER; REUTER, 1983; GRUENWALD *et al.*, 1996; MOHAN *et al.*, 1992).

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. Nonetheless, those activities can introduce a costly overhead to the execution of new transactions, since the MMDBs is not able to schedule them until the recovery process has finished (MOHAN; LEVINE, 1992; GRUENWALD *et al.*, 1996; MALVIYA *et al.*, 2014).

In most MMDBs, the recovery process is performed offline. In other words, the database system becomes available to service new transactions only after the full recovery process is completed. One may claim that MMDBs may keep database replicas to ensure high availability. Nevertheless, the replication mechanism is not immune to errors and unpredictable defects in software and firmware. Besides, adding high availability infrastructure to a system can become expensive to deploy and maintain (WU *et al.*, 2017; FAERBER *et al.*, 2017; TAN *et al.*, 2015).

1.2 The proposed solution

In this sense, this thesis proposes an instant recovery mechanism for MMDBs, denoted ***MM-DIRECT*** **(Main Memory Database Instant RECovery with Tuple consistent checkpoint)**. *MM-DIRECT* implements a generic recovery mechanism, which may be embeded in any main memory database system. *MM-DIRECT* allows MMDBs to schedule new transac-

tions immediately after the system starts up. Figure 2 emphasizes the benefits of instant recovery (described in this work) concerning MMDB standard recovery (implemented by most MMDBs). Figure 2 represents the average transaction throughput over small time intervals. Looking more closely at Figure 2, one may observe that the standard recovery mechanism (represented by the yellow line) has downtime after a failure while the database is recovering. On the other hand, the instant recovery mechanism (blue line) is able to schedule transactions immediately after the system starts up during the recovery process. Thus, applications and users do not notice the recovery process, giving the impression that the system was instantly restored. Since the proposed instant recovery mechanism schedules transactions as quickly as possible, it delivers higher Input/Output Operations per Second (IOPS) rates, i.e., it executes workloads faster than the standard recovery mechanism.

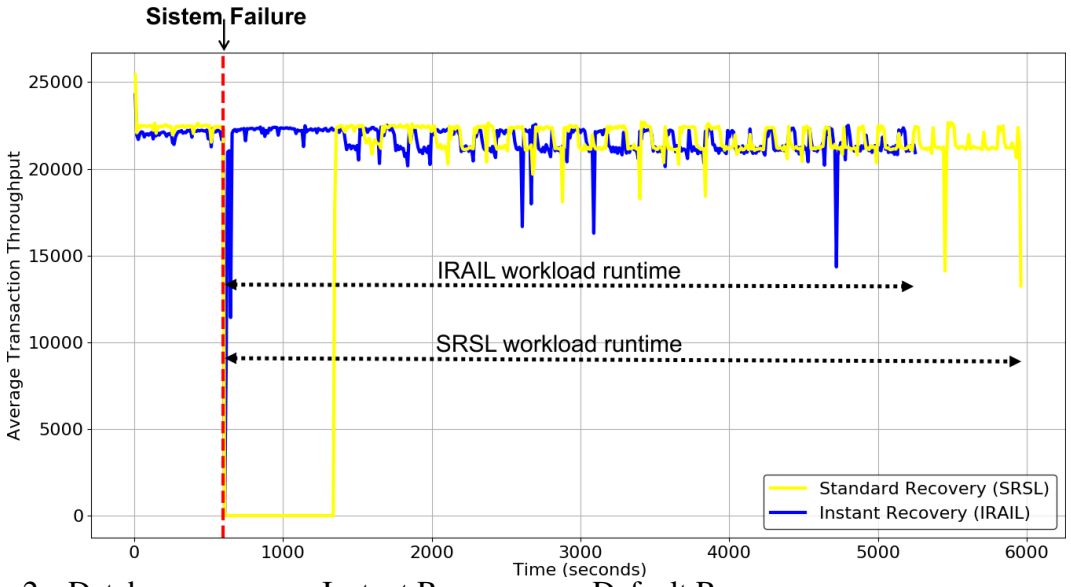


Figure 2 – Database recovery: Instant Recovery vs. Default Recovery

The main idea behind the concept of instant recovery is to recover database tuples individually, tuple by tuple incrementally. This mechanism naturally supports high availability because transactions can access tuples immediately after they are restored into memory. Also, during recovery, if transactions require tuples that have not yet been loaded into memory, those tuples can be recovered on demand. Thus, this recovery mechanism allows the system to recover the database completely in parallel with transaction processing, as a tuple can be used for new transactions as soon as it is restored to memory. (SAUER *et al.*, 2017; MAGALHÃES *et al.*, 2021; MAGALHÃES, 2021).

Conventional database systems write update records into a sequential log file. For

that reason, the recovery process in such systems cannot be performed incrementally and on-demand, i.e., it makes instant recovery impossible. The sequential log file has inefficient reads for individual log records. A full log scan must be done to restore a given tuple individually. In this scenario, transactions can only be executed after the recovery process has finished (SAUER *et al.*, 2017; MAGALHÃES *et al.*, 2021; MAGALHÃES, 2021).

The instant recovery mechanism proposes a new log file organization. In this case, the log file is structured as a B^+-tree (COMER, 1979) to enable the efficient retrieval of log records individually (SAUER *et al.*, 2017; MAGALHÃES *et al.*, 2021; MAGALHÃES, 2021). The proposed B^+-tree log file uses the database's tuple ID as a search key value. Update records for a tuple are stored in the leaf node that contains a search key value equal to the tuple ID. This log organization enables the efficient recovery of a single tuple. A single fetch on the B^+-tree can retrieve all necessary log records to recover one tuple. Thus, the system can recover the database by traversing the leaf nodes of the B^+-tree, restoring tuple by tuple incrementally. If necessary, a tuple can be restored on-demand by fetching the B^+-tree using the tuple ID. Although this mechanism has been designed to use a B^+-tree, the mechanism also enables to support other index structures, such as a Hash table (LITWIN, 1980).

The recovery mechanism proposed in this work stores transaction update records in a B^+-tree during system running. However, transactions do not wait for log record insertions in the B^+-tree log for them to be committed. The proposed logging technique writes log records to a sequential log file at commit time, which is much faster than writing records into the B^+-tree log file. Log records are written from the sequential log file to the B^+-tree in the background. Such a strategy does not impact transaction processing throughput. In addition, the B^+-tree log file uses a buffering mechanism to mitigate the rate of writing records between the sequential log and the B^+-tree.

The log is a fast-growing file, mainly on OLTP-style systems, characterized by many update operations on the database. Therefore, log truncation must be performed periodically to reduce the number of log records to be processed during recovery and consequently reduce recovery time. In this sense, the proposed logging mechanism implements a tuple consistent checkpoint to reduce the number of log records stored in the B^+-tree log file. Furthermore, the proposed technique does not lose the work done if the checkpoint process is interrupted, unlike other checkpoint techniques.

We have empirically evaluated the proposed instant recovery mechanism in a proto-

type implemented on the Redis (REDIS LABS, 2020a; REDIS, 2020), an in-memory database, in order to prove its efficiency and suitability for MMDBs. Redis database was chosen due to its simple architecture and easy-to-understand source code. Redis is a key-value database. However, the proposed logging mechanism can be in any other database, such as relational databases. The database and workloads used in the experiments correspond to those from Memtier benchmark (MEMTIER BENCHMARK, 2020). Memtier tool has been employed in several works, such as Behravesh *et al.* (2019), Zhang e Swanson (2015), Cao *et al.* (2016), Ouaknine *et al.* (2017), Magalhães *et al.* (2021), Kim e Lee (2021), Ibrahim *et al.* (2018), Choi *et al.* (2018), and Zhang *et al.* (2021).

1.2.1 Research hypotheses

In this thesis we supposed the following research hypotheses:

- **H1**: An indexed log file is required to support incremental and on-demand database recovery.
- **H2**: Inserts of records into the indexed log asynchronous to transaction commit are necessary to avoid decreasing of transaction throughput during normal MMDB function.

The experiments shown in section 5 were defined to confirm those hypotheses about the proposed recovery mechanism.

1.3 Thesis contributions

The main contributions of this thesis are the following:

- A recovery mechanism for MMDBs that allows to schedule new transactions during the recovery process, giving the impression that the system was instantly restored. The mechanism restores database tuples incrementally and on-demand.
- A very lightweight logging technique that can efficiently read/write log records in order to restore tuples individually without degrading transaction processing, or as little as possible.
- A checkpoint technique that propagates log record actions to reduce the recovery time. The technique is also able to checkpoint the most frequently used tuples in order to interfere as little as possible in the system's performance. Besides, the work done by this technique persists even though the checkpoint process is not completed.
- We analyze the behavior of the proposed instant recovery mechanism through experiments

in an OLTP workload. The results showed that, although the mechanism retrieves data from secondary memory to recover a database after a system failure, the system maintains high throughput rates during the recovery process and normal database processing.

1.4 Thesis publications

1. We produced and presented the work "Performance Tuning in NoSQL Databases" through the initial research we did for the doctorate. This work was presented at the "3rd Regional School of Informatics of Piauí" (MAGALHÃES *et al.*, 2017).
2. We produced and presented the work "In-Memory Database Management Systems" at the "14th Brazilian Symposium on Information Systems" (MAGALHÃES *et al.*, 2018b).
3. We produced and presented the work "Big Data Management and Processing with In-Memory Databases" at the "1st Latin American Computer Update Journey" that was a satellite event of the "44th Latin American Computer Conference" (MAGALHÃES *et al.*, 2018a).
4. We presented our PhD project plans (Main Memory Databases Instant Recovery) at the "Database Theses and Dissertations Workshop" in the "34th Brazilian Symposium on Databases" (MAGALHÃES *et al.*, 2019).
5. Despite the efforts and publications, database recovery is not well understood by the research community. In-memory database systems are not taught sufficiently in database courses or discussed enough in database textbooks. Thus, we produced a survey (Main Memory Database Recovery: a Survey) to elucidate the main issues regarding in-memory database recovery. The survey was published in ACM Computing Surveys (MAGALHÃES *et al.*, 2021).
6. We developed a prototype in Redis database to evaluate the feasibility of the indexed log for in-memory database instant recovery proposed in this thesis. The experiment results performed using the prototype were presented and published in a paper (Indexed Log File: Towards Main Memory Database Instant Recovery) in the "24th International Conference on Extending Database Technology (EDBT 2021)" (MAGALHÃES *et al.*, 2021). When this work was published, the prototype had implemented only the techniques of logging and instant recovery after failures. The checkpoint had not yet been implemented.
7. We presented our PhD project plans (Main Memory Databases Instant Recovery) at the "VLDB PhD Workshop" in the "47th International Conference on Very Large Data Bases

(VLDB 2021)" (MAGALHÃES, 2021).

8. We presented the work "Main Memory Database Recovery: a Survey" that we published in ACM Computing Surveys (item 5) as "Distinguished Published Paper" at the "36th Brazilian Symposium on Databases" (SBBD2021, 2021).

1.5 Thesis organization

The remainder of this thesis is organized as follows. A thorough discussion about classical database recovery mechanisms is provided in Chapter 2, focusing on ARIES-style recovery, a quite popular algorithm for recovering traditional databases. Thereafter, that chapter sheds light on different strategies implemented by MMDBs, some technologies that boosted the development of MMDBs, and the techniques for implementing recovery in MMDBs. Chapter 3 discusses the related work. It describes some log-structured recovery techniques and the main features of recovery mechanisms delivered by well-know MMDBs. Chapter 4 presents the proposed mechanism for MMDBs instant recovery. Chapter 5 discusses the results of empirical experiments. Chapter 6 presents proposals for futures works. Finally, Chapter 7 concludes this work.

2 BACKGROUND

2.1 Database recovery overview

This section aims to present the main concepts of DBMS recovery focusing on disk-resident systems. We briefly review the ACID properties and some buffer replacement protocols, features of database crashes, recovery methods, and our main discussion, the ARIES recovery algorithm. Furthermore, this section discusses some variant ARIES algorithms, and high availability techniques.

A transaction is a set of database reads/writes that must be handled as a single and indivisible unit. A standard example of a transaction is the bank money transfer. A bank transfer is a method of transferring funds from one bank account to another. The implementation of a transfer basically consists of debiting an amount of money from one account and depositing the same amount in the other account. If one of the transfer operations fails, all operations must be undone, as if they had never happened. To achieve this indivisibility, a database system must maintain the Atomicity, Consistency, Isolation, and Durability (ACID) properties (HÄRDER; REUTER, 1983).

A brief explanation of ACID properties follows below (HÄRDER; REUTER, 1983).

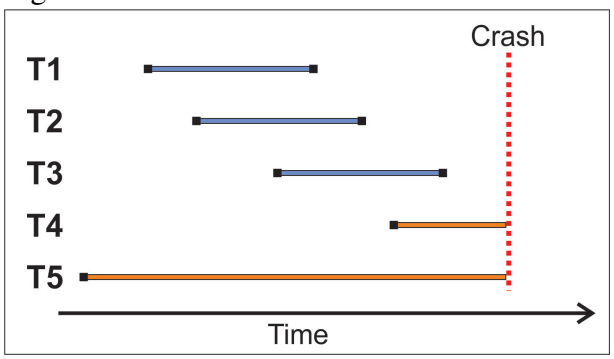
- **Atomicity**: A transaction is indivisible, i.e., all transaction operations must be performed properly on the database, or none of them.
- **Consistency**: A transaction executing in isolation must maintain database consistency.
- **Isolation**: Each transaction does not notice other transactions performed even in concurrent systems.
- **Durability**: Transaction updates must persist in the database after commitment.

Atomicity and durability of transactions are ensured for the recovery manager component. After a crash (e.g., power cut, software bugs, storage errors, and others), the database can be in an incorrect state. At this moment, the recovery manager must ensure that unfinished transactions will not have their actions reflected in the database (atomicity); and completed transactions will have their modifications written in the database, even if they have not been flushed to secondary memory (durability). Such situations may exist due to buffer manager priority of deciding when to evict a database page from the buffer pool. Both phenomena may imply an inconsistent database state if a recovery mechanism is not adopted (HÄRDER; REUTER, 1983; MOHAN *et al.*, 1992).

For performance reasons, most DBMSs adopt Steal and No-force buffer manager page replacement policies. The buffer manager is the DBMS component that managers the database buffer pool and reads/writes pages from/to the secondary memory. The steal approach is used when the buffer manager protocol allows flushing dirty pages to secondary storage before the transaction commit. A page is called dirty when the page has some updates that are not yet written in the secondary storage. With steal, the system does not need a large buffer to store all required pages because pages of an unfinished transaction can be flushed to secondary storage to free up space in memory for new requested pages. If pages of committed transactions do not need to be flushed at commit time, the approach is called no-force. In no-force, updated pages of a committed transaction can remain in the memory when other transactions need to update these pages, reducing the Input/Output (I/O) cost to flush pages multiple times (HÄRDER; REUTER, 1983; MOHAN *et al.*, 1992).

Most recovery techniques do not interfere in buffer manager protocols, i.e., they support steal and no-force approaches. This scenario makes the database recovery cost more expensive. After a crash, the recovery manager must undo unfinished transaction actions written on secondary memory and redo committed transactions that did not flush to secondary storage (HÄRDER; REUTER, 1983; MOHAN *et al.*, 1992; EICH, 1986). Figure 3 illustrates a database failure. Transactions T1, T2, and T3 must persist on the database since they have committed before the crash. Even though T1, T2, and T3 have not been flushed to secondary storage before the crash, the recovery manager must ensure the durability of these transactions. Transactions T4 and T5 have not finished before the crash and, consequently, are incomplete. The updates of uncommitted transactions must be undone to preserve database consistency (HÄRDER; REUTER, 1983).

Figure 3 – A database crash scenario



Source: Härder e Reuter (1983).

2.1.1 Features of database crashes

There are many different types of crashes. Each type can affect database processing, such as software defects (e.g., bugs or wrong inputs), hardware defects (e.g., power failure or disk error), or human error (e.g., a wrong device mounted or something intentionally done). Each type of failure must be handled differently (HÄRDER; REUTER, 1983). Database crashes can consider three categories: transaction, system, and media failures.

A transaction can be seen as a unit of consistency and recovery. A crash of a single active transaction means that one of its operations can not be applied to the database and, consequently, the database consistency may be incorrect. Thus, any update by that transaction on the database should be canceled. This type of failure occurs more often than the other two, and thus an efficient recovery form is essential. The usual procedure for recovery after a transaction failure is a Transaction UNDO, i.e., transaction rollback. Transaction failure involves only software failure, with no loss of memory or media (HÄRDER; REUTER, 1983).

A system failure occurs whenever an event causes a loss of data in the main memory, such as hardware error, operating error, code error, and power cut. After a system crash, two types of undesirable transactions may exist: *(i)* uncommitted transactions whose effects have been flushed to secondary memory, denoted persistent unfinished transactions (or loser transactions), and *(ii)* committed transactions whose effects have not been written on secondary storage, denoted non-persistent committed transactions (or winner transactions). Such transactions may exist due to the steal and no-force buffer manager protocols. Thus, the effects of any persistent unfinished transaction should be undone (Global UNDO), and any non-persistent committed transaction has to be redone (Partial REDO). System failure is the result of memory loss without media loss (GRAY *et al.*, 1981; HÄRDER; REUTER, 1983; MOHAN *et al.*, 1992).

A media crash is a secondary memory failure, such as an Hard Disk (HD) error. These failures can break down parts of the database or the entire database. After a media failure, assuming media replacement, the system must redo all the transactions performed in the database (Global REDO); or to load the last database backup and redo the log operations forward the backup. In media failure, there is only media loss, without loss of memory and software failure (HÄRDER; REUTER, 1983). If the specific location of the media failure can be identified, a redo of only the affected area would be warranted. The authors in work (GRAEFE; KUNO, 2012) considered a single-page failure as the fourth class of database failure. That work handles failures restricted to a page set of a storage device. They proposed a single-page recovery algorithm to

repair online individual pages instead of recovering the whole media device. The single-page recovery is detailed in Section 3.1.

2.1.2 Recovery method

Most database systems implement the Write-Ahead Logging (WAL) (MOHAN *et al.*, 1992) protocol for recovery purposes. The WAL uses in-place updating, i.e., the changes of some data must replace the page in the same location from which it was read on secondary storage. The WAL protocol ensures that, before an in-place update, information of every data change is written in a log archive on stable storage, even though both the steal and no-force approaches are adopted by DBMS. Stable storage is non-volatile storage. Thus, a DBMS stores log records for each transaction update for recovery purposes. The log file should be copied to different disks and in different locations to survive crashes. The log is a growing sequential file, in which every record has a Log Sequence Number (LSN) that is a unique identifier assigned in ascending sequence. After a page update, the LSN corresponding to that update is stored in a field called pageLSN in the header of that page. This field allows the system to know if a log record was the last to update a page. The system writes log records for each of the following update actions:

- **Update**: An update record is written before modifying data.
- **Commit**: A commit record is written at commit time.
- **Abort**: An abort record is stored if a transaction aborts.
- **End of Transaction (EOT)**: When a transaction finishes (aborts or commits), some additional actions must be taken. An end record is written when these actions are executed.

Below, we describe the main fields of log records implemented in most recovery techniques (MOHAN *et al.*, 1992):

- **LSN**: An identifier that should be assigned in a monotonically increasing order. LSN can be the logical address of its log record. Version numbers or timestamps can also be values for LSN.
- **Type**: Indicates the record type (e.g., update, commit, abort, or end).
- **TransID**: Transaction's identifier that generated the log record.
- **PrevLSN**: LSN of a transaction's previous log record. This field links the log records of a transaction. Thus, it is possible to repeat the transaction updates in descending order.
- **PageID**: The page identifier in which the log record applied updates.

– **Update**: This information depends on the log abstraction level and can be *(i)* the data after and/or before the update performed (e.g., tuple or page), or *(ii)* the operation performed in a data (e.g., update, insert, or delete commands).

The log information can be recorded at different abstraction levels, such as: physical, logical, and physiological. Physical logging records the state of the database updated, and only physical units are stored (e.g., page number, offset, and length). Logical logging records the state transition of the database and contains operation descriptions in a higher-level (e.g., insertion of a record in a table). In Physiological logging, the log records identify a page physically but record the update performed logically as an operation (MOHAN *et al.*, 1992; WU *et al.*, 2017; MALVIYA *et al.*, 2014).

Logical logging tends to be faster than physical logging during transaction processing. Usually, logical logging stores fewer items on the log than physical logging, to record modifications. However, logical logging is slower at system restarting because it must re-execute the transactions. Physical logging only needs to copy the data without any processing in the data. Logical logging requires deterministic transactions. Otherwise, it can add complexity to the recovery (e.g., date() and time() are non-deterministic functions). Physical logging needs more storage than logical logging, to record the after and before images of an update. Typically, disk-resident databases perform physical logging, and in-memory databases prefer logical logging (WU *et al.*, 2017; MALVIYA *et al.*, 2014). However, for performance reasons, commercial disk-based DBMSs, such as MySQL (MYSQL, 2020) and Oracle (ORACLE, 2020), often implement physiological logging. In a physiological log, a record refers to a page and logical operations on the page. For example, for each update, only its before image is logged together with the operation to redo the update (MOHAN *et al.*, 1992; TUCKER, 2004; GRAY; REUTER, 1993).

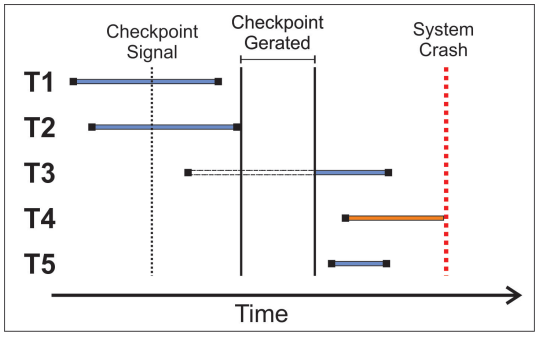
Some DBMSs, such as System R (GRAY *et al.*, 1981) and DB2/MVS V1 (CRUS, 1984), first perform the Undo pass and then the Redo pass. This approach is called selective redo paradigm. According to Haerder et al. (HÄRDER; REUTER, 1983), logging is tied to concurrency control in that the granule of logging determined the granule of locking. If the DBMS applies page logging, it must not use locking granularity less than pages. ARIES solves this problem through the repeating history paradigm. This paradigm performs the Redo phase before the Undo phase in order to support fine-granularity locking regardless of logging granularity (HÄRDER; REUTER, 1983; MOHAN *et al.*, 1992).

The logging schema ensures the persistence of data during transaction processing. However, log truncation must be performed because the log is a growing file and secondary storage has limited space. Therefore, the checkpoint is needed to create a new good point from which database recovery can begin to apply the changes contained in the log after a failure. Furthermore, the checkpoint is needed to speed up recovery, since fewer log records need to be applied during recovery. There are some checkpoint techniques, such as Transaction-Consistent Checkpoint (TCC), Action-Consistent Checkpoint (ACC), and Fuzzy Checkpoint (HÄRDER; REUTER, 1983; MOHAN *et al.*, 1992).

TCC technique creates a database state where no update transaction was active. before the checkpoint process begins, uncommitted transactions must be completed, and new transactions must not be initiated. When the last transaction update is completed, all dirty pages are flushed to secondary storage and a checkpoint record is stored in the log file. After a crash, the recovery manager can disregard the log records before the checkpoint since all modifications before the checkpoint were flushed to the secondary storage. Thus, the log record only needs to be redone starting after the checkpoint record. However, TCC degrades the system performance because the creation of transactions is stopped while the checkpoint process is executed (HÄRDER; REUTER, 1983; LIN *et al.*, 1997; KIM *et al.*, 2012).

Figure 4 illustrates the TCC technique. When the checkpoint signal arrives (i.e., when a checkpoint should start), the checkpoint does not allow the new transaction T3 to be executed and waits for the active transactions T1 and T2 to end. Then, the checkpoint flushes all transaction updates to secondary memory. When the checkpoint process is completed, the new transactions T3, T4, and T5 can preform. After the crash, the system should redo T3 and T5, and roll back T4. T1 and T2 were flushed to secondary storage before the checkpoint and must not be redone or undone (HÄRDER; REUTER, 1983).

Figure 4 – A Transaction-Consistent Checkpoint scenario

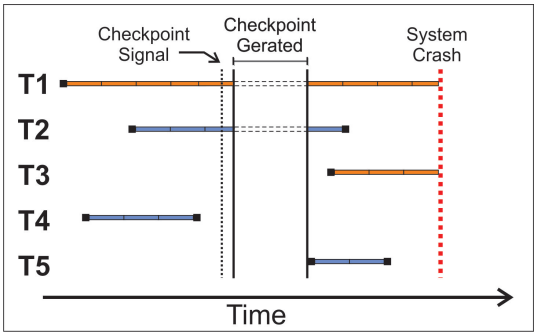


Source: Härder e Reuter (1983).

The ACC approach pauses transaction processing to be generated. ACC puts the system into quiescence on the transaction actions level; flushes all active transaction update actions; and, finally, writes a checkpoint record in the log file. When a crash occurs, the recovery manager knows that active transaction updates before the last checkpoint were written to the secondary storage. Thus, the recovery manager does not need to redo the transaction updates before the last checkpoint and can only process the log from checkpoint record. ACC causes performance degradation because it pauses transaction performing during the checkpoint process (HÄRDER; REUTER, 1983; LIN *et al.*, 1997).

An ACC scenario is shown in Figure 5. When the checkpoint signal arrives, the system pauses the transaction processing, i.e., it pauses the execution of the active transactions T1 and T2. Then, the system flushes all transaction updates to secondary memory. When the checkpoint is completed, the transactions T1 and T2 can continue performing, and the new transactions T3 and T5 can start executing. After the crash, T1 and T3 must be undone, and T5 must be redone. T2 updates after the checkpoint must be redone. T2 updates before the checkpoint were flushed to secondary memory and do not need to be redone. All changes from T4 are part of the checkpoint and, consequently, nothing needs to be done about T4 (HÄRDER; REUTER, 1983).

Figure 5 – A Action-Consistent Checkpoint scenario



Source: Härder e Reuter (1983).

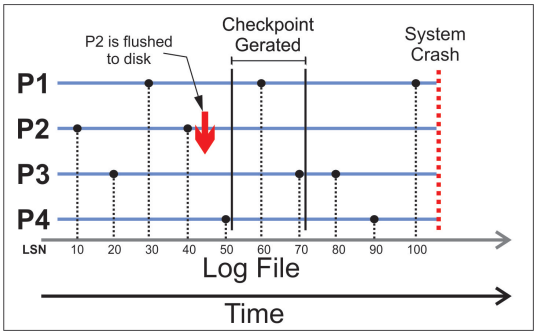
Both TCC and ACC require interference in transaction processing to get a state of database consistency. In contrast to those checkpoints, Fuzzy Checkpoint avoids propagation activities to reduce checkpoint costs. Instead of writing pages in secondary memory, the fuzzy approach stores information about buffer occupation: dirty pages, and active transactions (their states and the addresses of their most recently written log records). This technique does not stop the transaction performing, i.e., the checkpoint is generated during the execution of transactions.

When the Fuzzy process finishes, a checkpoint record is written on the log file (HÄRDER; REUTER, 1983; MOHAN *et al.*, 1992; LIN *et al.*, 1997).

After a crash, the system reads checkpoint information and log records forward from the last checkpoint record to identify the dirty pages and active transactions at crash time. The transactions identified as active are redone. Then, unfinished transactions are undone. Section 2.1.3.2 explains in more details the database recovery by fuzzy checkpoint. The fuzzy is cheaper than TCC and ACC because it requires fewer write operations. However, it makes the recovery more expensive since log records should be processed before the redo and undo phases (HÄRDER; REUTER, 1983; MOHAN *et al.*, 1992; LIN *et al.*, 1997).

Figure 6 shows a fuzzy checkpoint scenario. The pages P1, P2, P3, and P4 have modifications written on log file, represented by dots associated with an LSN. For example, the page P1 has logged update data on records with LSNs 30, 60, and 100. During checkpoint performing, the checkpoint records information about dirty pages P1, P3, and P4. Information about P2 is disregarded since modifications in this page were flushed to secondary memory before the checkpoint. The checkpoint must also record information about active transactions that were not represented in this example. After the crash, P1, P3, and P4 are identified as dirty pages. Moreover, the active transactions are also identified. That tree dirty pages and the active transactions are redone. Then, the unfinished transactions are undone (HÄRDER; REUTER, 1983; MOHAN *et al.*, 1992).

Figure 6 – A Fuzzy Checkpoint scenario



2.1.3 ARIES algorithm

Algorithms for Recovery and Isolation Exploiting Semantics (ARIES) (MOHAN *et al.*, 1992) was designed to support steal and no-force buffer approaches, fine-granularity, WAL, total and partial rollbacks, and fuzzy checkpoint. The main aim of this section is to provide a

brief overview of the ARIES recovery method. The following subsections discuss how ARIES performs logging and recovery.

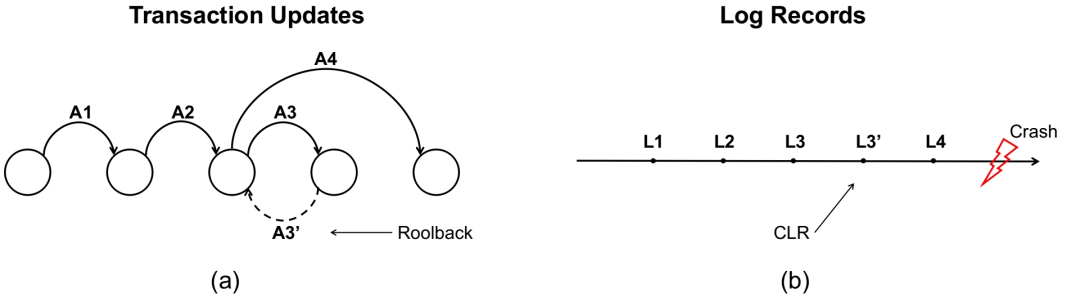
2.1.3.1 Logging

ARIES uses WAL to ensure atomicity and durability properties of transactions. In addition to earlier recovery methods, ARIES introduced a Compensation Log Record (CLR). CLRs describe actions performed during a rollback (partial or total) in normal or recovery processing. Additionally, CLRs are redo-only log records. This type of record is an update that will never be undone, i.e., the recovery manager never undoes an undo action from rollbacks. A CLR record has a UndoNxtLSN field that points to the predecessor of the undone log record. The process of undoing a transaction starts from its last record and ends at the first one. Thus, UndoNxtLSN is similar to the PrevLSN field (MOHAN *et al.*, 1992; MOHAN, 1999). CLRs ensure a bounded amount of logging during rollbacks, even if there are successive failures during recovery or of nested rollbacks. In contrast to ARIES, some systems, such as AS/400 (CLARK; CORRIGAN, 1989), DB2/MVS V1 (CHENG *et al.*, 1984), and NonStop SQL (GAWLICK *et al.*, 1987), undo the same CLR one or more times in face of multiple failures.

Figure 7 shows an example of a CLR generated by a partial rollback. Figure 7 (a) illustrates updates of a transaction T1 and Figure 7 (b) represents the log records generated by T1 actions. The log records L1, L2, L3, L3', and L4 were generated by the updates A1, A2, A3, A3', and A4, respectively. A3' is the undo action of A3 and the remaining T1 actions are normal updates. Thus, only L3' is a CLR. If the log record L3 represents the deletion of the row R1 on page P1, the CLR L3' stored during the undo of L3 would describe the insertion of R1 on P1. As a result, the state of the page P1 is always viewed as forward on the log, even when some original updates are being undone. Besides, the UndoNxtLSN field of L3', which is the CLR for log record L3, points to log record L2, which is the predecessor of L3. During the crash recovery, assuming that T1 is a loser transaction, T1 updates must be undone starting from the log record L4. When the recovery manager reaches L3', instead of undoing L3' and then L3, it skips to log record L2 (MOHAN *et al.*, 1992; MOHAN, 1999).

CLRs describe actions to undo transaction updates. However, these actions do not need to be the inverse of that update. ARIES allows logical undo. For example, B^+-tree update operations, in ARIES/IM (MOHAN; LEVINE, 1992) and ARIES/KVL (MOHAN, 1990), are not the exact inverse of their undo operations. This is very efficient during recovery. Furthermore,

Figure 7 – CLR generated by a partial rollback



Source: Mohan *et al.* (1992).

ARIES supports operation logging. For example, increment/decrement operations store log records that represent the quantity of increment/decrement in a field rather than before and after field values. This feature supports semantically-rich modes of locking and, consequently, allows multiple transactions to concurrently update the same data (MOHAN *et al.*, 1992; MOHAN, 1999).

ARIES implemented the concept of Nested Top Action (NTA). An NTA starts inside another action (parent). However, an NTA has no special relationship with its parent. For example, an NTA can not read an atomic object updated by its parent, and the NTA commitment is not relative to its parent. NTA changes are committed regardless of whether the parent changes are later committed or not (LISKOV; SCHEIFLER, 1983). For instance, index management is an example of an NTA application. Page split/delete in indexes needs the atomicity property independent of transactions. ARIES implements NTAs by dummy CLR records. A dummy CLR is written at the end of NTA performing. The UndoNxtLSN field of the dummy CLR points to the most recent log record of its parent transaction stored before the start of the NTA. After a failure, the NTAs changes are redone, if necessary. Moreover, if the dummy CLR is not stored before the failure, the NTA changes are undone (MOHAN *et al.*, 1992; ROTHERMEL; MOHAN, 1989; MOHAN, 1999).

2.1.3.2 Restart recovery

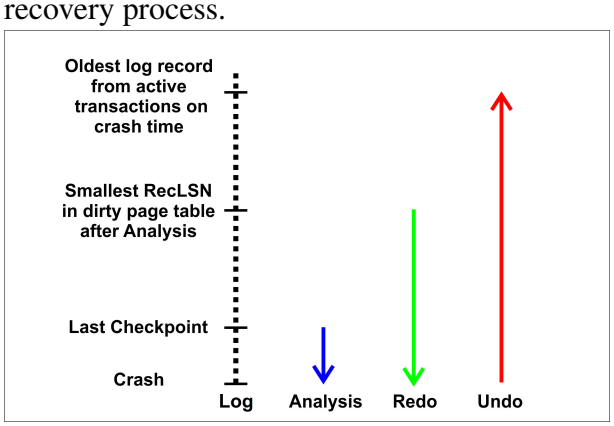
ARIES recovery schema consists of three phases (analysis, redo, and undo). We summarize the ARIES phases below and Figure 8 illustrates the execution of these phases over the log (MOHAN *et al.*, 1992; MOHAN, 1999).

1. **Analysis**: Identifies dirty pages and active transactions before the crash, and the log record to start the Redo.
2. **Redo**: Repeats transaction updates to recover the system to the same state it was during

the crash time.

3. **Undo**: Undoes the updates of unfinished transactions to recover the system to the last consistent state before the crash.

Figure 8 – ARIES phases over the log during the recovery process.



Source: Ramakrishnan e Gehrke (2003).

When the system restarts, the recovery manager examines the most recent checkpoint to initialize the transaction table and dirty page table with checkpoint information. The transaction table tracks the state of active transactions. The important fields of the transaction table are: *(1)* TransID, transaction ID; *(2)* LastLSN, the LSN of the latest log record written by the transaction; and *(3)* UndoNxtLSN, the LSN of the next record to be processed during a rollback. The dirty page table represents information about dirty buffer pages. This table consists of two fields: *(1)* PageID, page ID; and *(2)* RecLSN (recovery LSN), the LSN of the latest log record that updated the page (ROTHERMEL; MOHAN, 1989; MOHAN *et al.*, 1992; MOHAN; LEVINE, 1992; MOHAN, 1999).

After finishing to examine the checkpoint, the analysis phase reads the log forward from the checkpoint record to find all active transactions and dirty pages until the crash time. When the Analysis phase ends, the active transaction and dirty page tables lost in the crash are rebuilt in memory. In addition, this phase calculates the log record to start the Redo phase by locating the minimum RecLSN value in the dirty page table, i.e., the oldest log record of the dirty pages (ROTHERMEL; MOHAN, 1989; MOHAN *et al.*, 1992; MOHAN; LEVINE, 1992; MOHAN, 1999).

Redo phase scans forward the log file from the record given by the Analysis to reapply the updates of transactions that had dirty pages before the system failure. This process is called Repeating History Paradigm. An update from a log record must be redone when the

pageLSN from the page modified by this record is less than the LSN of the current record, i.e., the page content is out of date. When the log record is redone, the pageLSN field is updated to the LSN of the redone log record. This repeating history process rebuilds the database to the same state it was before the crash time. If a transaction was undone before the crash (e.g., a transaction abort), the CLR records must be reapplied (ROTHERMEL; MOHAN, 1989; MOHAN *et al.*, 1992; MOHAN; LEVINE, 1992; MOHAN, 1999).

The Undo phase is responsible for rollback updates of loser transactions in reverse log order. The system continuously processes the log record with the maximum LSN value between the log records not yet undone from all the loser transactions, until no loser transaction remains to be undone. For each log record to process, a CLR record is stored in the log and the corresponding action is undone if the log record represents a modification. If the log record is a CLR then no action is undone. After a log record of a transaction is processed, the next record to be processed for that transaction is determined by the PrevLSN or the UndoNxtLSN field of the record, if the record is a nonCLR or a CLR, respectively (ROTHERMEL; MOHAN, 1989; MOHAN *et al.*, 1992; MOHAN; LEVINE, 1992; MOHAN, 1999).

2.1.4 Variant ARIES algorithms

This section exposes the main aspects of some algorithms that extend the original ARIES algorithm: ARIES for Nested Transactions, ARIES with Restricted Repeating of History, ARIES for Index Management, ARIES for Linear Hashing with Separators, and ARIES for the Client-Server Architecture.

2.1.4.1 ARIES for nested transactions

The original ARIES algorithm was designed to work with single-level transactions. ARIES for Nested Transactions (ARIES/NT) (ROTHERMEL; MOHAN, 1989) extends the original ARIES algorithm implementing a simple and efficient recovery method for a very general model of nested transactions. A nested transaction (sub-transaction or child transaction) is a database transaction that is started within the scope of a transaction that has already started (parent transaction). Nested transaction updates are visible for unrelated transactions only after the parent of the sub-transaction is committed (MADRIA, 1997). The ARIES/NT mechanism allows partial rollbacks of sub-transactions, concurrent execution of ancestor and descendent sub-transactions, and upward and downward inheritance of lock (ROTHERMEL; MOHAN,

1989; MOHAN, 1999).

The ARIES log uses the PrevLSN field to create a Backward chain (BW-chain) of records. BW-chain describes the updates of a transaction in descending order. In ARIES/NT, BW-chains from child and parent transactions are linked. A c-committed record is logged when sub-transaction commits (so-called committed inferior). The c-committed record has a pointer that links the parent's BW-chain to the last record of the BW-chain of its committed inferior. The BW-chain of a transaction linked to the BW-chains of its committed inferiors forms a Backward chain tree (BWC-tree). Additionally, the CLR's UndoNxtLSN field is slightly different from the one in the original ARIES. Instead of storing a single LSN, UndoNxtLSN contains a set of LSNs, and each of them points to the next log record from a transaction or committed inferior to be processed during undo. During recovery, the Redo step performs similarly to ARIES. However, the Analysis and Undo steps have been modified to support BWC-trees (ROTHERMEL; MOHAN, 1989; MOHAN, 1999).

2.1.4.2 ARIES with restricted repeating of history

ARIES with Restricted Repeating of History (ARIES-RRH) (MOHAN; PIRAHESH, 1991) is a modified version of ARIES that intends to handle the Redo phase more efficiently. The goal of the ARIES repeating history paradigm is to reapply the updates of transactions that had dirty pages before the system failure. The repeating history is necessary to support fine-granularity locking. However, this approach can redo transactions that will be undone in the Undo step. This wasted work delays the availability of the system, i.e., it delays the time to process new transactions. In contrast with repeating history, the selective redo paradigm (discussed in Section 2.1.2) performs the Undo phase before the Redo phase, but it does not support fine-granularity locking (MOHAN; PIRAHESH, 1991; MOHAN, 1999).

ARIES-RRH combines the repeating history paradigm with the selective redo paradigm when fine-granularity locking is not required. Besides, it considers that not all unapplied but logged updates need to be redone, even if fine granularity is required for the data. ARIES-RRH can reduce the number of I/Os and the amount of Central Process Unit (CPU) processing during the recovery, but it still maintains all the good features of the original ARIES algorithm (MOHAN; PIRAHESH, 1991; MOHAN, 1999).

2.1.4.3 ARIES for index management

ARIES for Index Management (ARIES/IM) (MOHAN; LEVINE, 1992) is a method based on ARIES for controlling concurrency and persisting recoverable B^+-tree indexes. The main goal of ARIES/IM was to improve the algorithms for index concurrency control of the original System R (GRAY *et al.*, 1981) in order to enhance its performance, concurrency, and functionalities. It allows structure modification operations (i.e., page split/deletion), tree operations (i.e., key insert/delete/fetch and range scan), and high concurrency during tree traversals. The object of locking in ARIES/IM is the individual index entry (data-only locking). Besides, the concurrency method was improved to avoid transaction deadlocks during its rollback. The main idea of the ARIES/IM algorithm is to update the leaves and propagate the tree if overflow/underflow changes occur. Thereby, few sub-trees are locked during operations. This provides a very high concurrency level (MOHAN; LEVINE, 1992; MOHAN, 1999; RODEH, 2008).

It is worth mentioning that a previous work called ARIES using Key-Value Locking (ARIES/KVL) (MOHAN, 1990) addressed concurrency control and recovery of B-tree indexes. However, the concurrency enhancements of ARIES/KVL were still considered inadequate. The author of the paper, Mohan (1990), discussed only the possibility of correct logging and recovery in the face of concurrent executions. The paper did not implement logging and recovery in ARIES/KVL. Works before ARIES/IM and KVL researched recovery with concurrency control (e.g., (BAYER; SCHKOLNICK, 1977), (MINOURA, 1984), (SAGIV, 1986), (SHASHA, 1985), and (SHASHA; GOODMAN, 1988)) but they did not consider concurrency control and B-trees indexes with fine-granularity locking in face of failures.

2.1.4.4 ARIES for linear hashing with separators

Linear Hashing with Separators (LHS) (LARSON, 1988) is a dynamic hashing schema in which any record can be retrieved in the file in only one access to secondary memory, given its corresponding key value. Very little has been discussed in the literature about the impact of concurrent transactions accessing hash structures. ARIES for Linear Hashing with Separators (ARIES/LHS) (MOHAN, 1993) is an adaptation of the original ARIES that handles the concurrency control and recovery with LHS. ARIES/LHS exploits the concepts of NTAs and logical undo to provide efficient recovery and high concurrency. Dealing with deadlocks

during rollbacks has been a known issue since System R. ARIES algorithms try to avoid rollback of transactions to avoid deadlocks. ARIES/LHS implements recovery techniques to handle transactions and prevents them from deadlock while they are being undone. ARIES/LHS has not been implemented (MOHAN, 1993; MOHAN, 1999).

2.1.4.5 ARIES for the client-server architecture

ARIES for the Client-Server Architecture (ARIES/CSA) (MOHAN; NARANG, 1994) is a recovery algorithm designed to work in Client-Server (CS) architectures. In a CS environment, the database is managed only by the server, and clients cache pages requested to the server in local buffer pools. In ARIES/CSA, when a client modifies a page on its buffer, it produces log records for that update. However, the client can not store the records in a local log file on secondary storage. Each client generates LSNs for its records and stores them in its buffer. At appropriate times, the client sends its records to the server that stores them in a single log file. In addition to managing log records received from different clients, the server takes care of global locking and recovery. ARIES/CSA supports the same features of the original ARIES: WAL, fine-granularity locking, partial rollbacks and steal and no-force buffer management policies. ARIES/CSA has not been implemented (MOHAN; NARANG, 1994; MOHAN, 1999).

2.2 Techniques for high availability

In the face of failures, the high availability ensures that the system remains operational with close to zero downtime. High availability infrastructure incurs more complexity and cost for a system. Different applications may have different availability needs. For example, a mission-critical system needs 100% availability while many other systems should need a simpler availability approach. High availability can be achieved through replication: updates are propagated to replicas (WIESMANN *et al.*, 2000; ZAMANIAN *et al.*, 2019). According to Gray et al. (GRAY *et al.*, 1996), replication protocols can be either lazy or eager.

Lazy replication propagates transaction updates to replicas only after the commitment. This method has a minimal overhead but it has the possibility of data loss. Lazy schema is acceptable on systems where strong consistency is not crucial. It is very difficult to solve the inconsistencies created by lazy replication. These inconsistencies can be eliminated by an eager approach. Eager replication ensures that the transaction updates reach the replicas

before the transaction is considered committed. This schema leads to a strongly consistent replication. Since each replica has an identical copy of the database, it is easier to handle failures. However, efficient eager replication protocols are more difficult to design (GRAY *et al.*, 1996; WIESMANN *et al.*, 2000; ZAMANIAN *et al.*, 2019).

Paxos (LAMPORT, 2001; LAMPORT, 2019), ZooKeeper Atomic Broadcast (ZAB) (JUNQUEIRA *et al.*, 2011), and Raft (ONGARO; OUSTERHOUT, 2014) are consensus protocols, which can be applied for state machine replication. Thus, they provide high availability and fault tolerance in distributed systems. A consensus protocol allows a set of distributed processes to be coordinated as a coherent group even in the presence of several faulty processes. Many distributed systems use state machine replication to synchronize data among the replicas. Disk mirroring (BITTON; GRAY, 1988) is another example of high availability technique. Mirroring data is referred to as storage replication. Disk mirroring is the replication of logical storage device volumes onto separate physical storage devices in real-time. From a DBMS perspective, it is important to note that an event in which a failed disk in a redundant array is automatically repaired cannot be considered a media failure (SAUER *et al.*, 2017). For this reason, replication is not covered in this work.

2.3 In-memory database overview

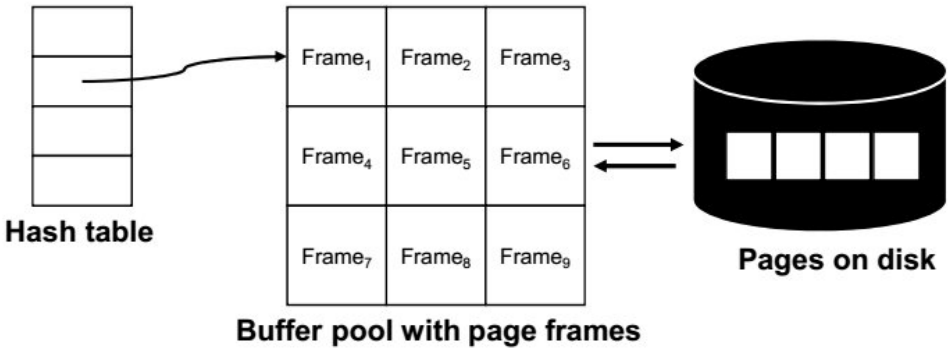
This section very briefly discusses the architectural choices and implementation techniques for MMDBs. The purpose of this section is to provide minimal knowledge about MMDB design for the next sections. MMDBs avoid the traditional design of disk-resident databases for performance reasons. The design approaches for MMDBs are diverse. 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. In addition, this section highlights the core hardware technologies that boosted MMDBs (ZHANG *et al.*, 2015; FAERBER *et al.*, 2017).

2.3.1 Data storage

Disk-resident databases use a buffer pool to access records. The buffer pool handles pages from secondary storage to main memory, and vice versa. The buffer pool is an shared array of page frames, whose pages are the same size as the database pages (e.g., 8KB or larger

is typical). Figure 9 depicts a buffer pool schema. Most buffer pool implementations use a hash table to access pages in the buffer and map them to their location on disk. When a page is requested, the storage manager looks for the page in the hash table. If the page is not in buffer, the storage engine performs disk I/O to bring the page into memory. When the page is in the buffer, the system must still perform an indirect action to access the data. The buffer pool indirection consists of two basic steps: *(i)* accessing the page in the main memory, and *(ii)* calculating the offset within the page to reach the record (HELLERSTEIN *et al.*, 2007; FAERBER *et al.*, 2017).

Figure 9 – Buffer pool schema.



Source: Faerber *et al.* (2017).

MMDBs avoid the page-based indirection through a buffer pool. MMDB are not dependent on disk logical layout (e.g., page size). Thereby, they can store data in any format in order to achieve high I/O rates. MMDBs commonly use pointers for direct access to records in memory. Pointers can save space for large values that appear several times in a database since the data can be stored just once and referenced by memory pointers. Moreover, avoiding page indirection, the system needs fewer CPU cycles to access the data in memory (LARSON; LEVANDOSKI, 2016; HAZENBERG; HEMMINGA, 2011; PUCHERAL *et al.*, 1990; FAERBER *et al.*, 2017).

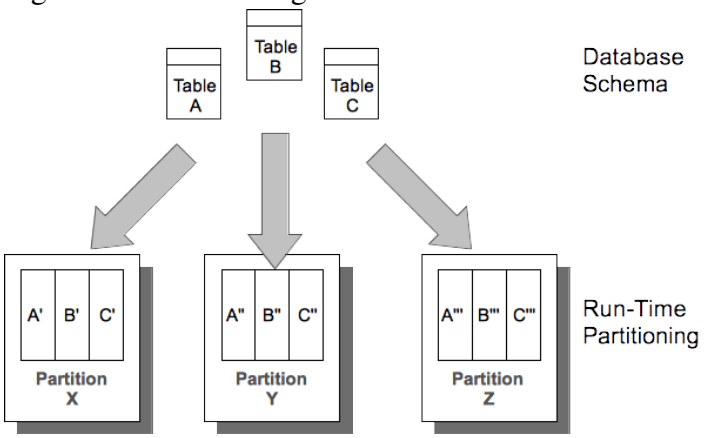
Physical pointers, instead of buffer pool indirection, can improve access to records in memory in order of magnitude for MMDBs. The work (LEHMAN *et al.*, 1992) evidenced this improvement. This work compares IBM Starburst's memory-resident storage component with its default disk-oriented storage component. The results showed that the in-memory component can perform significantly better than the disk-oriented component, even when both systems have all data cached in memory. In many cases, the experiments showed that the buffer pool manager was responsible for about 40% of the total execution time of a query.

Some MMDBs physically partition the database (e.g., H-Store, VoltDB, and Calvin

(THOMSON *et al.*, 2012)). In partitioned systems, a transaction is performed serially in a partition by a single-thread execution engine and has exclusive access to the data at this partition. Besides, it enables a transaction to use the available resources alone in its partitions while it is performing (e.g., a CPU core). A database can be partitioned on different machines. Thus, the database can be larger than the memory available of a single node. On the other hand, no-partitioned in-memory systems (e.g., Hekaton, SAP HANA, MemSQL (CHEN *et al.*, 2016), and Oracle TimesTen) can access any record in the database. This approach prevents the balance of partitions accessed frequently (TAFT *et al.*, 2014; FAERBER *et al.*, 2017).

Figure 10 is an example of partitioning tables. Each partition can reside in different sites creating a distributed database. Multiple queries can run in parallel at different partitions because each site operates independently. For example, table A is divided exclusively into parts A', A" and A"' which are stored in partitions X, Y and Z, respectively. While one query is scanning A' in partition X, another one can scan A" in partition Y at the same time. A query that needs to scan table A entirely needs to acquire exclusive access to partitions X, Y, and Z. The database design must consider concurrent query access to different partitions (VOLDB DOCUMENTATION, 2020).

Figure 10 – Partitioning table scheme.



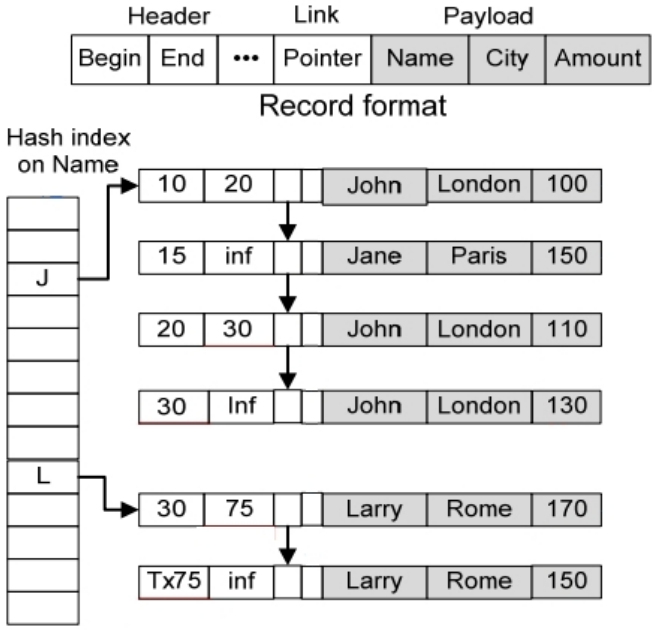
Source: VolDB Documentation (2020).

While disk-resident databases typically implement in-place updating, many MMDBs choose multi-versioning of data (e.g., Hekaton, HyPer, and SAP HANA). In multi-versioning, the updated version of the data is written to a different location from the previous version of the data (shadow version). Multi-versioning leads to the easier implementation of non-blocking concurrency control protocols. Blocking protocols have more context switch than non-blocking protocols. The context switching is a source of overhead for MMDBs. Session

2.3.2 explains MMDB concurrency control in more detail. Another advantage of multi-versioning is to enable snapshots. Snapshot is equivalent to a materialized database state in an instant of time. Snapshots are useful for durability and recovery purposes. MMDB can generate snapshots by an asynchronized consistent checkpoint, i.e., the system does not need to pause transaction processing to create a snapshot (CHEN *et al.*, 2011a; MALVIYA *et al.*, 2014; NEUMANN *et al.*, 2015). Session 2.5.3 discusses checkpoint techniques for MMDBs.

Figure 11 illustrates a data multi-versioning scheme. This example represents a simple bank account table containing six versions of records. The table has the user-defined columns Name, City, and Quantity. In addition to the table fields, a version record also includes the Link (Pointer) field from a hash index and a header that defines version's valid time by the Begin and End fields. Hash bucket J contains four records, but only two records are valid versions. John's current version (John, Londom, 130) has a valid time from 30 to infinity. This version was created by a transaction that committed at time 30 and has no end time, i.e., it is still valid. John's oldest version (John, London, 100) was valid from time 10 to time 20 when John's account was updated. The obsolete versions should be discarded by a garbage collector. Usually, the checkpoint component acts as a garbage collector (DIACONU *et al.*, 2013).

Figure 11 – Data multi-versioning scheme.



Source: Diaconu *et al.* (2013).

An organizational choice is whether the database is row or column-oriented (ABADI *et al.*, 2008). The row format employs the *N*-ary Storage Model (NSM) whose attribute values for

a single tuple are stored contiguously. NSM is a good choice for Online Transaction Processing (OLTP) workloads. OLTP transaction queries tend to handle an individual entity at a time and most (if not all) of its attributes. In the example in Figure 12, all tuple attributes with ID 101 are stored one after another, followed by all tuple attributes with ID 102 (COPELAND; KHOSHAFIAN, 1985; ARULRAJ *et al.*, 2016a).

Figure 12 – OLTP-oriented N-ary Storage Model.

ID	IMAGE-ID	NAME	PRICE	DATA
101	201	ITEM-101	10	DATA-101
102	202	ITEM-102	20	DATA-102
103	203	ITEM-103	30	DATA-103
104	204	ITEM-104	40	DATA-104

Source: Arulraj *et al.* (2016a).

The columnar format uses the Decomposition Storage Model (DSM) whose tuples' values for a single attribute are stored contiguously. DSM is employed in on-line Online Transaction Processing (OLAP) systems. OLAP queries tend to operate multiple entities at the same time. Besides, they typically access only a subset of the attributes for each entity. Figure 13 shows an example of DSM storage layout. The DBMS stores all the values of the first attribute (ID) contiguously, followed by those that belong to the second attribute (IMAGE-ID) (COPELAND; KHOSHAFIAN, 1985; ARULRAJ *et al.*, 2016a).

Figure 13 – OLAP-oriented Decomposition Storage Model.

ID	IMAGE-ID	NAME	PRICE	DATA
101	201	ITEM-101	10	DATA-101
102	202	ITEM-102	20	DATA-102
103	203	ITEM-103	30	DATA-103
104	204	ITEM-104	40	DATA-104

Source: Arulraj *et al.* (2016a).

OLTP and OLAP workloads are executed in different databases: transactional database and data warehouse, respectively. However, some systems need to transform the most up-to-date data into critical insights. These systems should use the Flexible Storage Model (FSM) that generalizes the NSM and DSM models and can support Transaction/Analytical Processing (HTAP) workloads. FSM stores the frequently accessed data in an NSM layout. When a particular item of data becomes little (if ever) accessed, the system reorganizes that data into a DSM layout. However, synchronizing data from NSM to DSM can add overhead to the

system (COPELAND; KHOSHAFIAN, 1985; ARULRAJ *et al.*, 2016a). SAP HANA explores a tuple propagation schema from row to columnar storage to support HTAP (SIKKA *et al.*, 2012). HyPer uses a virtual memory schema to perform OLTP and OLAP transactions on the same database (FUNKE *et al.*, 2014).

2.3.2 Concurrency control

Management database concurrency control is an extremely important performance topic that has been researched since the early days of relational databases (e.g., (GRAY *et al.*, 1975), (BERNSTEIN *et al.*, 1987), (GRAY; REUTER, 1993), (BERNSTEIN; GOODMAN, 1981), (WEIKUM; VOSSEN, 2002), (HOLANDA *et al.*, 2008), (MONTEIRO *et al.*, 2009), (YU *et al.*, 2014), and (BAILIS *et al.*, 2014)). Several researches compared the tradeoffs between pessimistic and optimistic concurrency control protocols, such as (CAREY; STONEBRAKER, 1984), (AGRAWAL; DEWITT, 1985), (TAY *et al.*, 1985), (AGRAWAL *et al.*, 1987), and (FRANASZEK; ROBINSON, 1985). Pessimistic Concurrency Control (PCC) assumes that concurrency conflicts will occur frequently and tries to avoid them during the performing, blocking, and aborting transactions (WEIKUM; VOSSEN, 2002). Optimistic Concurrency Control (OCC) assumes that multiple transactions can be completed without concurrency conflicts until the commitment when a validation step checks conflicts. If conflicts happened, transactions can be aborted. This method does not handle locks (KUNG; ROBINSON, 1981).

For performance reasons, MMDBs avoid implementing a lock manager. Lock manager is a component to manage the transaction locking requests. Before accessing the data, the transaction must exchange messages with the lock manager to lock/unlock that data. This indirection is a common approach for disk-resident databases since secondary storage I/Os are the major source of overhead (BERNSTEIN *et al.*, 1987; WEIKUM; VOSSEN, 2002). However, the lock manager mechanism is a performance bottleneck for MMDBs. A PCC approach for MMDBs is to embed metadata into records to control access to them (HAZENBERG; HEMMINGA, 2011; ZHANG *et al.*, 2015). Garcia-Molina e Salem (1992) propose a "lock bit" added to each object to indicate whether the object is blocked or not. In work (REN *et al.*, 2012), the authors suggest integer counters preceding the record value, which represents the number of transactions requesting locks on the record.

Many MMDBs implement Multi-Version Concurrency Control (MVCC). In MVCC, reader transactions do not have to block data, since writer transactions perform their modifications

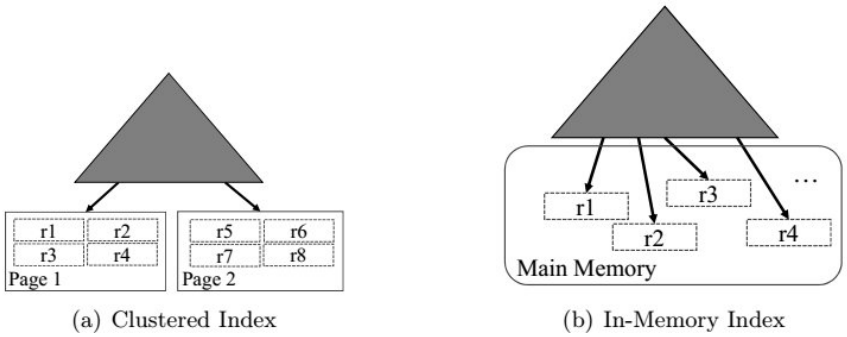
on different versions of the data rather than readers. However, MVCC has the overhead of creating new data versions during updates and periodically removing obsolete versions. Some MMDBs implement an optimistic MVCC, such as Hekaton and HyPer. An optimistic MVCC is cheaper than handling locks. In addition, it allows the scalability of cores, since transactions can perform unlocked until the commitment without context switch (NEUMANN *et al.*, 2015; LOMET *et al.*, 2012; FAERBER *et al.*, 2017). Agrawal et al. (AGRAWAL *et al.*, 1987) suggest that an optimistic concurrency control is a good approach when conflict rates are low. However, it degrades the system performance when many transactions need to roll back and restart due to high conflict rates. SAP HANA is an example of MMDB that implements a pessimist MVCC. SAP HANA uses MVCC to ensure consistent read operations. However, it uses exclusive write locks at record-level. A transaction must obtain a write lock on the record before updating it. Another transaction that needs to update that record has to wait until the lock is released (LEE *et al.*, 2013; ZHANG *et al.*, 2015).

H-Store, VoltDB, and Calvin are examples of MMDBs that implement the partitioning approach. In this method, a transaction is performed serially in a partition, i.e., a transaction performs exclusively on the necessary partitions. Each partition is assigned to a different core. Thus, transactions can be performed in parallel on different partitions. A transaction can only start to perform if all necessary partitions are available. Otherwise, the transaction will be aborted and restarted as soon as its partitions are available. This approach avoids complex concurrency protocols to support concurrent thread updates (KALLMAN *et al.*, 2008; THOMSON *et al.*, 2012; STONEBRAKER; WEISBERG, 2013).

2.3.3 Indexing

Although in-memory systems have high throughput rates compared with disk-based systems, they still need efficient indexing to speed up data retrieval in databases. Indexes allow the system to find data quickly, without scanning the entire database (Guo; Hu, 2010). MMDB indexes were designed to care about memory and cache utilization rather than secondary memory access, as in disk-based databases. Besides, MMDBs do not implement buffer-pool indirection (as discussed in Section 2.3.1). Therefore, indexes can store direct pointers to records, instead of primary keys (AILAMAKI *et al.*, 1999). Figure 14(a) presents a disk-based clustered index, whose indexes point to pages that store records. On the other hand, Figure 14(b) shows a MMDB indexing approach whose indexes have pointers directly to records (FAERBER *et al.*, 2017).

Figure 14 – Differences in disk-based and in-memory index organization.



(a) Clustered Index (b) In-Memory Index

Source: Faerber *et al.* (2017).

Rosenblum *et al.* (1995) showed that ignoring disk I/O, the main memory can stall the processor for over 50% of the execution time due to cache misses. For this reason, MMDB indexing research tries to optimize cache line usage. The cache line can be considered the unit of transfer between memory and processor. Height Optimized Trie (HOT) (BINNA *et al.*, 2018), Cache Sensitive Search Trees (CSS$^+$-Trees) (RAO; ROSS, 2000), Cache Sensitive B$^+$-Trees (CSB$^+$-Trees) (RAO; ROSS, 2000), Prefetching B$^+$-Trees (pB$^+$-Trees) (CHEN *et al.*, 2001), Fast Architecture Sensitive Tree (FAST) (KIM *et al.*, 2010), and Adaptive Radix Tree (ART) (LEIS *et al.*, 2013) are indexing techniques for cache efficiency. Hash indexes also can be further optimized for better cache utilization (FAN *et al.*, 2013).

Currently, it is common that multi-core machines have more than one thousand cores. MMDBs can run on CPUs with a staggering amount of parallelism (TAN *et al.*, 2015). Some index strategies consider multi-core parallelism. Physiological Partitioning (PLP) design applies logical-only partitioning and uses Multi-rooted B$^+$-Tree (MRBTree) index to ensure latch-free accesses to indexes on the database partitions (PANDIS *et al.*, 2011). Bw-tree (LEVANDOSKI *et al.*, 2013b) and Mass-Tree (MAO *et al.*, 2012) are concurrent indexing techniques that apply latch-free by atomic CPU primitives operations to update index state. Examples of other indexing techniques for MMDBs: Optimistic Lock Coupling (OLC) (LEIS *et al.*, 2019; ROSENBLUM; OUSTERHOUT, 1992), T-Tree (LEHMAN; CAREY, 1986), Δ-Tree (CUI *et al.*, 2003), and BD-Tree(CUI *et al.*, 2004).

2.3.4 Query processing

MMDBs avoid the traditional iterator model (Volcano-style processing) (GRAEFE; MCKENNA, 1993). This model is commonly implemented in disk-resident databases since it facilitates the combination of arbitrary operators. Furthermore, it generates a huge number

of function calls (e.g., next()) which results in evicting the register contents. However, the indirection from next() functions and poor code locality from iterator generic nature can add overhead to MMDBs (HAZENBERG; HEMMINGA, 2011; ZHANG *et al.*, 2015; FAERBER *et al.*, 2017). MMDBs compile queries directly to machine code to avoid interpretation and parsing overhead. Compiled code can make better use of memory and CPU. On the other hand, it needs to know all transactions in advance (DIACONU *et al.*, 2013; KEMPER; NEUMANN, 2011; MENON *et al.*, 2017). For example, a transaction must be a single stored procedure in VoltDB (MALVIYA *et al.*, 2014; STONEBRAKER; WEISBERG, 2013).

2.3.5 Durability and recovery

Although MMDBs store the database in the main memory, the database must also be written in stable storage for persistence and recovery purposes. Durability and recovery activities are the only reason that in-memory systems access secondary memory since memory RAM is volatile. MMDBs avoid implementing ARIES-style recovery for performance reasons. They try to reduce log amount by performing a redo-only logical log, i.e., the log file stores only redo operations at a higher level. Commonly, MMDBs generate snapshots periodically to speed up the restart process. MMDBs restore by loading the last snapshot and then replaying the log forward from the checkpoint record. Section 2.5 explains in more detail the main memory database system recovery.

2.4 Core technologies for in-memory systems

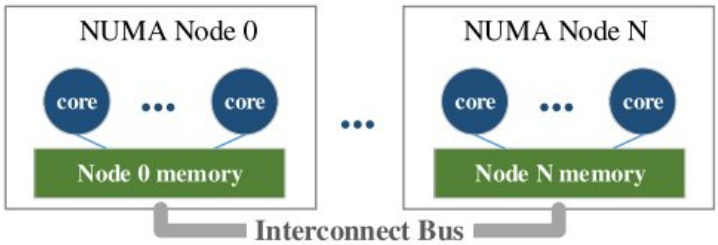
In-memory database systems have been studied since the 1980s. However, the high price and limitations of the hardware in that period made MMDBs infeasible. The emergence of new technologies boosted interests in MMDBs. Hardware/architecture solutions have been increasingly exploited for performance gain. These improvements offer promising alternatives for in-memory systems to reach their full potential (TAN *et al.*, 2015; ZHANG *et al.*, 2015). This section presents the core technologies that leverage MMDBs.

2.4.1 Non-uniform memory access

The CPU speed grew faster than the main memory speed in the last decades (NYBERG *et al.*, 1994). Non-Uniform Memory Access (NUMA) architecture allows addressing the

data starvation problem in modern CPUs. In a NUMA system, each processor can access a local main memory with minimal latency, and a remote main memory with longer latency, as shown in Figure 15. In addition to improving the main memory bandwidth, this technique can increase the total memory size of the system. NUMA allows creating memory domains by clustering multiple memory controllers in one node (MAAS *et al.*, 2013; LEIS *et al.*, 2014).

Figure 15 – NUMA topology.



Source: Zhang *et al.* (2015).

In the context of DBMS research, NUMA-awareness cares about minimizing the accesses to remote NUMA domains by data partitions (e.g., (BORAL *et al.*, 1990), (DEWITT *et al.*, 1990), (ALBUTIU *et al.*, 2012), (MAAS *et al.*, 2013), and (LEIS *et al.*, 2014)); managing system memory accesses on latency-sensitive workloads (e.g., OLTP workloads) since NUMA has heterogeneous access latency (e.g., (POROBIC *et al.*, 2012), (POROBIC *et al.*, 2014)); and transferring the data across NUMA domains efficiently, i.e., data shuffling (e.g., (LI *et al.*, 2013)).

2.4.2 Hardware transactional memory

Transactional Memory (TM) provides a concurrency control method analogous to atomic database transactions. Although TM simplifies the implementation of DBMSs, it never took hold. Recently, Hardware Transactional Memory (HTM) has attracted new attention in database researches. The basic idea of HTM is to use the CPU cache as a local transaction buffer and provide isolation. In addition, transaction conflicts are detected by cache coherency. This mechanism leads to a very low-overhead transaction performing. However, HTM has drawbacks: transaction size limited to cache size, unexpected transaction abort due to false conflicts; and transaction duration limit due to events that can abort the transaction, such as context switches, interrupts or page faults (HERLIHY; MOSS, 1993; KARNAGEL *et al.*, 2014; LEIS *et al.*, 2014; MAKRESHANSKI *et al.*, 2015). HyPeR (LEIS *et al.*, 2014) and DBX (WANG *et al.*, 2014) databases explore HTM to achieve concurrency. The works (KARNAGEL *et al.*, 2014),

(MAKRESHANSKI *et al.*, 2015), and (WANG *et al.*, 2014) improve in-memory database index performance with HTM.

2.4.3 Non-volatile memory

Advanced Non-Volatile Memory (NVM) technologies include Phase-Change Memory (PCM) (RAOUX *et al.*, 2008), Memristors (STRUKOV *et al.*, 2008), and Spin-Transfer Torque Magnetic RAM (STTMRAM) (DRISKILL-SMITH, 2010). NVM is an emerging technology that promises the best properties from hard disks and DRAM: byte addressability, persistence with high performance, and large storage capacity. However, NVM has disadvantages, such as limited endurance, write/read asymmetry, and uncertainty of ordering and atomicity (DEBRABANT *et al.*, 2014; ARULRAJ *et al.*, 2015; ARULRAJ; PAVLO, 2017). Although there are many researches that describe the DBMS architecture affected by NVM (e.g., (RENEN *et al.*, 2019), (RENEN *et al.*, 2018), (OUKID *et al.*, 2017), (ARULRAJ; PAVLO, 2017), (HARRIS, 2016), (ARULRAJ *et al.*, 2016b), (GARG *et al.*, 2015), (ARULRAJ *et al.*, 2015) (SCHWALB *et al.*, 2015), (ZHANG *et al.*, 2015), (CHEN *et al.*, 2011b)), it is not clear how best to design a DBMS through NVM.

The existing architectures for disk and memory-oriented DBMSs are inappropriate for NVM. Disk and memory-oriented systems have an architecture whose design is based on the propagation of transaction updates from memory to secondary storage for durability purposes. The byte-addressable, low latency and durability features of NVM can remove that propagation costs (DEBRABANT *et al.*, 2014; ARULRAJ *et al.*, 2015; ARULRAJ; PAVLO, 2017). Section 2.5.5 very briefly discusses the recovery in NVM-resident database systems.

2.4.4 Single instruction multiple data

Single Instruction Multiple Data (SIMD) is an instruction set available on current processors. These instructions provide an easier alternative to achieve data-level parallelism in order to speed up the processing free from concurrency issues. A single SIMD instruction is performed in parallel on several data points. Each instruction operates multiple data objects. However, SIMD limits the maximum parallelism allowed and has constraints on data structures to operate (WILLHALM *et al.*, 2009; NEUMANN, 2011). An efficient MMDB design should consider data-level parallelism. SIMD instructions can speed up expensive database operations (e.g., join and sort) (CHHUGANI *et al.*, 2008; NEUMANN, 2011). SAP HANA implements a

vector schema by SIMD to speed up dictionary decompression during scans (WILLHALM *et al.*, 2013). SIMD can improve vector-style computation commonly applied in Big Data analytics (RAMAN *et al.*, 2008; MÜHLBAUER *et al.*, 2013).

2.4.5 Remote direct memory access

Remote Direct Memory Access (RDMA) network allows a machine to read/write from/to a pre-registered memory region of another machine, without involving the kernel and CPU on the remote side. In contrast to traditional design, in an RDMA network, a server does not coordinate a request from the client. A client can access the server's memory directly, without any server actuation. RDMA has zero CPU overhead compared to Ethernet message passing. However, RDMA has limits in synchronizing multiple accesses and inefficient coordination of access to remote memory by different machines. Besides, RDMA can not easily connect with the traditional Ethernet directly. With less CPU overhead, a database server can rely on wimpier cores and achieve the same or better performance (MITCHELL *et al.*, 2013). FaRM implements lock-free reads over RDMA (DRAGOJEVIC *et al.*, 2014). Hyper can generate backup files in stable storage via RDMA to relieve the server CPU of the data transmission task (KEMPER; NEUMANN, 2011).

2.5 In-memory database recovery

Recovery activities (logging, checkpoint, and restart) are the only way to recover an MMDB to the last consistent state before a crash. Systems can keep database copies for higher availability. However, high-availability infrastructures are not immune to some sources of failures that can cause multiple and shared failures. Moreover, hardware improvements can lead to a significant cost to the database infrastructure. Thus, recovery techniques are necessary to avoid failures and repair failed systems as quickly as possible. We briefly covered the disk-based database recovery approaches in Section 2.1, in which we discussed key database recovery concepts. Throughout this section, we will highlight how logging, checkpoint, and restart (recovery) processes are handled on MMDBs. Logging stores update records to stable storage (non-volatile memory) during transaction processing. Checkpoint creates a backup database (snapshot) from the main memory to stable storage. After a system failure occurs, the restart reloads the snapshot from the secondary storage into the main memory and then replays

the log file from the checkpoint record. In general, MMDB durability and recovery seem like Disk-Based Database System (DBDS). However, these systems are very different in several details. For example, ARIES-style recovery protocols are avoided in MMDBs for performance reasons (JAGADISH *et al.*, 1993; GRUENWALD *et al.*, 1996).

2.5.1 Features of crash recovery in MMDB

The crash of MMDBs can be categorized in transaction and system failures. The transaction crash in MMDBs is similar to the transaction crash in disk-resident databases. When transaction updates do not reach their goal or violate any database integrity rule (e.g., duplication of the primary key), the transaction must be aborted and rolled back, as if it had not started. During transaction rollback, all effects of this transaction must be removed from the database. However, in many MMDBs, transactions do not update the database before the commitment. Each transaction performs updates locally and other transactions can see its updates only when it commits. Therefore, the system should only discard the updates of aborted transactions. For example, in systems that implement multi-versing storage, when a transaction is in the process of updating its versions of data, only it can access them. Only after the commitment, the data versions become the current database versions visible for all transactions. On the other hand, if the transaction aborts, it is necessary to only discard its data versions. Transaction failure does not affect the running of the system nor the transactions not involved in this failure. In addition, there is no loss of memory or media (EICH, 1986; Tang Yanjun; Luo Wen-hua, 2010).

When a system crash occurs in an MMDB, the system stops running and, consequently, the database in the memory is lost. The MMDB recovery after a system failure is quite different from the disk-resident database. The recovery process must perform two tasks: *(i)* reloading the database from the snapshot archive on secondary memory and *(ii)* replay the log actions also from secondary memory. After the restart, the system is recovered to the last consistent state before the failure. Most MMDBs perform redo-only logging. Thus, these systems do not undo transaction updates. There is not a media failure in MMDBs since the primary database resides on the main memory. The media to which the checkpoint and log files are written can fail, but this would not necessarily make the system stop running. This way, the system will no longer have guarantees of recovery from failures. However, if the system fails along with the media, assuming there are multiple checkpoints and log devices after the media is replaced, the failure can be treated as a system failure (EICH, 1986; LI; EICH, 1993; Tang

Yanjun; Luo Wen-hua, 2010).

2.5.2 Logging

Since the main memory is volatile, transaction modification actions must be stored on a log archive on stable storage during a transaction commitment. The need for a log on secondary memory can reduce the response time since the transactions must wait for a write to secondary storage before committing. Consequently, the log threatens system performance due to the secondary storage I/O (GARCIA-MOLINA; SALEM, 1992). Typically, in traditional disk-based databases, logging is not the main overhead problem since transaction processing is the main I/O overhead. However, logging may cause high I/O overhead depending on factors such as database size, memory available, and number and type of data updated by transactions. On the other hand, in an MMDB, commonly logging is the source of secondary memory I/Os during transaction processing (YAO *et al.*, 2016).

Logging in MMDBs is optimized to interfere as little as possible in the database transaction processing. MMDBs avoid physical logging because more log items are needed to record modifications. These systems try to reduce log volume. They prefer to perform logical logging, which generally stores fewer items on a log than physical logging. Moreover, MMDBs do not store before images of updates. They produce only the REDO log to reduce data flushed to secondary storage. Therefore, transactions must write log records only at the commitment. If the system fails, any update of uncommitted transactions will not persist. MMDBs often run OLTP workloads with short transactions. Each transaction can maintain an undo log locally in memory for aborting, if necessary. After the commitment, the undo log is discarded. Besides, most MMDBs prefer Solid-State Drive (SSD) as the log device to increase I/O performance. Several systems also avoid logging indexes. After a crash, indexes can be rebuilt in parallel to recovery (MALVIYA *et al.*, 2014; WU *et al.*, 2017; ZHENG *et al.*, 2014). For this reason, this work does not discuss index recovery.

To further minimizing the log traffic, some systems use a transaction-level (coarse-grained) logging technique called command logging (or transaction logging). In command logging, the transaction's logic is written to the log rather than the transaction's operations (operation logging). Each transaction must be a predefined stored procedure. The log records the stored procedure identifier of a transaction and its corresponding query parameters. Command logging is very lightweight, and it needs only one log record to store an entire transaction. This

technique generates a low overhead to transaction processing. However, it can slow down the recovery process because it needs to perform the transactions again (MALVIYA *et al.*, 2014; WU *et al.*, 2017).

MMDBs can use the group commit and pre-commit techniques to improve secondary memory access for writing log records. In group commit (HAGMANN, 1987; DEWITT *et al.*, 1984), the log records of several transactions performing at the same period are accumulated (e.g., up to the size of a page), and all records are written together in a single I/O operation to log file. This technique relieves the log bottleneck because it reduces the number I/Os to secondary storage since a single I/O flushes multiple transaction commit. Under the pre-commit technique (DEWITT *et al.*, 1984; GARCIA-MOLINA; SALEM, 1992), a transaction releases its locks as soon as its log records are sent to secondary memory. The transaction does not wait for the confirmation of writing log records to stable storage. Thus, a transaction can escape from the overhead of flushing log records. However, the transaction can lose the durability guarantee when failures happen.

2.5.3 Checkpoint

MMDB checkpoint is very different from the checkpoint in disk-resident database systems. The checkpoint process of some MMDBs propagates asynchronously transaction updates from the log file to a checkpoint archive in order to reduce recovery time and free up log space. The checkpoint materializes logical operations from the log file to the checkpoint archive. Instead of propagating log records, most MMDBs produce a consistent checkpoint commonly called snapshot. A snapshot is equivalent to a materialized database state in a specific instant of time. As soon as a checkpoint is performed, the checkpoint record is stored on the log. Checkpoints reduce recovery time since loading materialized data from the snapshot into memory is less costly than performing logical log operations. Moreover, in OLTP environments, commonly several log records can update the same data item (e.g., frequently updated tuples) (EICH, 1986; GARCIA-MOLINA; SALEM, 1992; DIACONU *et al.*, 2013; STONEBRAKER; WEISBERG, 2013).

Similar to logging, the checkpoint algorithm should not reduce significantly the transaction processing performance. Fuzzy checkpoint seems to be the best approach to databases. It does not pause the transaction processing and flushes less data than propagating log records or creating a snapshot. Yet, a fuzzy checkpoint depends on log records to undo and redo transactions

to restore the database after a failure. However, most MMDBs record only the REDO log. Several MMDBs prefer to perform an asynchronous transaction-consistent checkpoint. This technique creates a copy of the database state at system runtime without pausing transaction processing (SALEM; GARCIA-MOLINA, 1989; LIEDES; WOLSKI, 2006; REN *et al.*, 2016).

Many MMDBs perform the Copy-on-Update (COU) snapshot algorithm (EICH, 1986; DEWITT *et al.*, 1984). At the checkpoint's beginning, the system is put into COU mode. Then, the checkpoint scans all records in the database at the system runtime and copies them to a snapshot archive in the main memory. A bit-array is used to identify when a record was inserted, removed, or updated since the snapshot's beginning. The checkpoint skips inserted records. Before an update or delete, the original content of a record is copied to a shadow table so that the checkpoint can read the old version later. The old version is deleted as soon as it is scanned. A process serializes the snapshot from the memory to secondary storage until the checkpoint ends. COU has two overhead sources: locking and bulk bit-array resetting. COU must acquire locks to isolate the thread that writes the snapshot to the memory and the thread that flushes the snapshot to the disk. Moreover, COU must reset the bit-array before starting a new checkpoint period (CHEN *et al.*, 2011a; MALVIYA *et al.*, 2014). COU has many variants, such as in (SALLES *et al.*, 2009), (LIEDES; WOLSKI, 2006), and (FORK, 2020).

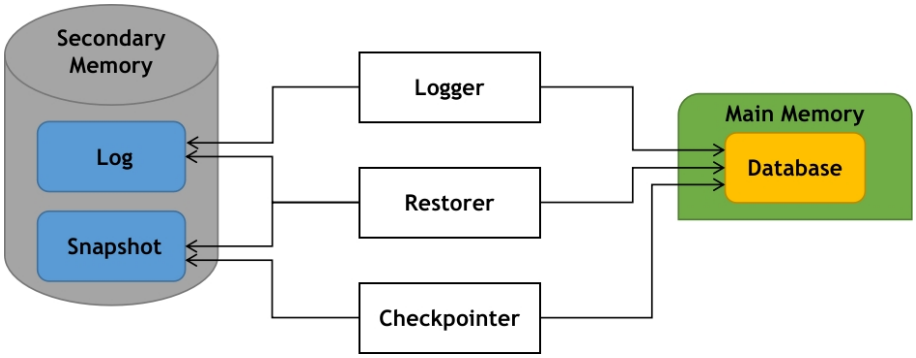
The COU technique may produce a consistent checkpoint during transaction processing. However, it can generate a memory usage overhead because it may potentially need an amount of memory twice the size of the database. The checkpoint can be considered as the most recent database backup because it brings to secondary memory the newer database copy. Thereby, the checkpoint eliminates the need for log entries before the last checkpoint record (CHEN *et al.*, 2011a; MALVIYA *et al.*, 2014). There are also other proposed snapshot algorithms such as Naive (BRONEVETSKY *et al.*, 2006; SCHROEDER; GIBSON, 2007), Zigzag (CHEN *et al.*, 2011a), PingPong (CHEN *et al.*, 2011a), Hourglass (LI *et al.*, 2018b; LI *et al.*, 2018a), and Piggyback (LI *et al.*, 2018b; LI *et al.*, 2018a).

2.5.4 Restart

Whenever a system crash occurs in an MMDB, the primary copy of the database is lost. In this case, the MMDB recovers the database by loading the last valid checkpoint. Thereafter, the recovery manager starts to execute the actions recorded in the logical log file forward from checkpoint record. The recovery component activities are briefly illustrated in

Figure 16. All update actions of the committed transactions are flushed to the log file on secondary memory by the Logger component. Periodically, the Checkpoint component produces a persistent file (snapshot) that reflects a consistent state of the database. 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 (GRUENWALD; EICH, 1991; GRUENWALD; EICH, 1994; WU *et al.*, 2017; FAERBER *et al.*, 2017).

Figure 16 – MMDB recovery component architecture.



Usually, MMDBs implements a simple algorithm to reload the snapshot called ordered reload. The ordered reload schema reloads the items in the same order in which they were written physically on the secondary memory. This schema can provide the fastest database reload. However, the system must load the entire snapshot archive into the main memory before starting the redo phase or bringing the database online (if there are no log records). It is possible that there are no records to replay in the log file, i.e., the snapshot is equal to the last database state before the system stops running. Before an intentional shutdown, the system stops the transaction processing and, then, performs a checkpoint to speed up the next system restart. However, this event can not be classified as a failure (GRUENWALD; EICH, 1991; GRUENWALD; EICH, 1994).

Several systems attempt to parallelize recovery as much as possible to achieve more and better performance. For example, Hekaton maintains multiple log storage devices to maximize I/O bandwidth during recovery (DIACONU *et al.*, 2013). SilioR implements a highly parallel log replay schema that prevents unnecessary allocations, copies, and work (ZHENG *et al.*, 2014). PACMAN (implemented in Peloton) enables parallel replay of transaction-level log records. It uses a graph of execution constraints among transactions to find parallelization opportunities (WU *et al.*, 2017). RAMCloud (ONGARO *et al.*, 2011) uses a log-structured approach (similar to log-structured file systems (ROSENBLUM; OUSTERHOUT, 1992)) to

allow fetching log and backup segments in parallel.

Many systems do not need the entire database to start running. According to the 80-20 rule, 80% of accesses to databases correspond to 20% of the items. For example, in OLTP workloads, generally few records are frequently accessed. When it is not necessary to load the entire database or the main memory is not large enough to fit the entire database, a partial load can be performed (LEVANDOSKI *et al.*, 2013a). For instance, the Anti-caching (DEBRABANT *et al.*, 2013) from H-Store and Siberia (ELDAWY *et al.*, 2014; LEVANDOSKI *et al.*, 2013a; ALEXIOU *et al.*, 2013) from Hekaton allow a database more extensive than available memory.

Achieving full performance in MMDBs for both recovery and transaction processing is a trade-off. Most MMDBs prefer to prioritize transaction processing over failure recovery. Most MMDBs perform a logical transaction-level logging and an asynchronous transaction-consistent checkpoint. These recovery techniques are very lightweight for transaction processing compared to ARIES-style techniques. On the other hand, they cause the MMDB recovery process to become slower than the ARIES recovery. With the advent of high-availability infrastructure, recovery speeds have become secondary in importance to run-time performance for most MMDBs. The reason for this is because high available infrastructure can continue to service transactions while a failed database is recovering. For example, database replicas can service transactions while the system is recovering after a failure (WU *et al.*, 2017; FAERBER *et al.*, 2017; MALVIYA *et al.*, 2014).

2.5.5 Database recovery in NVM-resident database systems

Recovery and logging processes in DBMSs running in NVM are quite different from those processes used in disk and main memory-resident DBMSs. The latter two systems propagate transaction updates from main memory to secondary storage for guaranteeing durability. Besides, the data are maintained in two copies (the database and the log) to ensure database recovery after a failure. Data propagation to secondary memory and duplication strategies employed in disk and main memory DBMSs would cause unnecessary performance degradation for databases stored in NVM (ARULRAJ *et al.*, 2016b; ARULRAJ; PAVLO, 2017).

Write-Behind Logging (WBL) (ARULRAJ *et al.*, 2016b) is a well-known recovery technique for DBMSs running in NVM. WBL enables the system to recover almost instantaneously from system failures. The main idea of the WBL is to store in the log which parts of the database were updated instead of how they were updated. The system writes changes to the

database before storing them in the log. Thus, the log is always slightly behind the contents of the database (ARULRAJ *et al.*, 2016b; ARULRAJ; PAVLO, 2017).

A Dirty Tuple Table (DTT) is used to track transaction updates. Each DTT entry has the transaction ID, the table modified, and a meta-data based on the write operation (insert, delete or update). The system uses the DTT information to abort a transaction, if necessary. Unlike the log and database that are stored in the NVM, the DTT is written to main memory. WBL uses a group commit interval to track the active transactions. All transactions committed before the current group commit interval are safely persisted on durable storage. When committing a group of transactions, the DBMS persists the changes contained in DTT and then records the final timestamp of the group commit interval in the log. In the case of a long-running transaction T consuming more time than the group commit interval, the system can also log T's commit timestamp (ARULRAJ *et al.*, 2016b; ARULRAJ; PAVLO, 2017).

When a system failure occurs, the DBMS ignores the effects of transactions that fall within the last group commit interval. As WBL was designed for multi-versing storage, ignoring an update only consists of not validating the updated version. Nevertheless, the authors also discuss how to adapt WBL for a single-version system. It is not necessary to redo transactions since the effects of all committed transactions are persisted (ARULRAJ *et al.*, 2016b; ARULRAJ; PAVLO, 2017).

The WBL technique does not need to periodically generate checkpoints to speed up recovery. This is because all transaction updates executed before the current group commit interval have already been persisted. Therefore, the log records stored before the current group commit interval can be removed safely. This allows the log file to be truncated to free up storage space (ARULRAJ *et al.*, 2016b; ARULRAJ; PAVLO, 2017).

In the context of database recovery, we can also mention some researches such as ETERNAL (GOMES *et al.*, 2019), NVRAM Group Commit (PELLEY *et al.*, 2013), Editable Atomic Writes (EAW) (COBURN *et al.*, 2013), NV-Logging (HUANG *et al.*, 2014), Fast Optimistic Engine for Data Unification Services (FOEDUS) (KIMURA, 2015), REWIND (CHATZISTERGIOU *et al.*, 2015), JUSTDO logging (IZRAELEVITZ *et al.*, 2016), and Hyrise-NV (SCHWALB *et al.*, 2016b). Other works related to database recovery in NVM are: Schwalb *et al.* (2016a), Arulraj *et al.* (2015), Pelley *et al.* (2013), Huang *et al.* (2014), Wang e Johnson (2014), Gao *et al.* (2011), and Agrawal e Jagadish (1989).

3 RELATED WORK

3.1 Log-structured recovery techniques

Although ARIES is a well-established recovery approach, recent hardware improvements have spurred the development of new software architectures that can better exploit modern hardware (GRAEFE; KUNO, 2012; SAUER, 2019). For example, flash storage promises higher performance than magnetic disks. However, it also presents its problems, such as limited endurance and write/read asymmetry (CAPPELLETTI *et al.*, 2013; BEZ *et al.*, 2003). The failure of individual pages in storage is described in (GRAEFE; KUNO, 2012) as the fourth class of database failures: single-page failure. This type of failure is not part of any of the traditional database failure classes. At worst, it could be considered a media failure, although only individual pages fail, not the entire device. The techniques presented here introduced the usage of log-structured file and write-optimized B-trees in the recovery process.

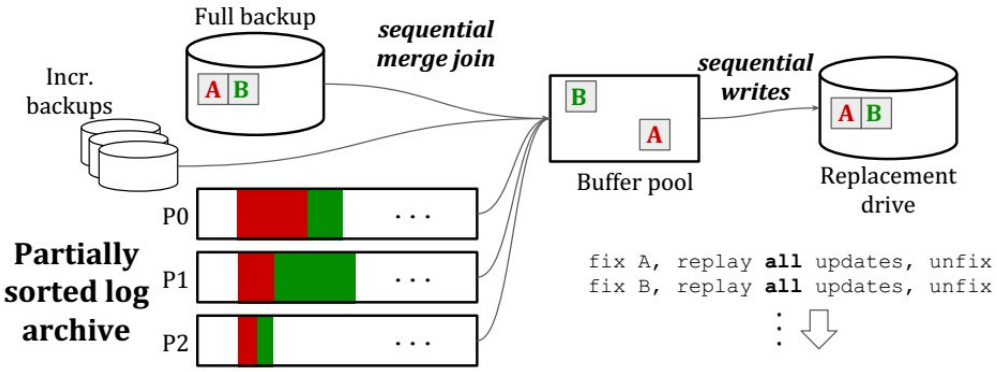
The Log-Structured Merge tree (LSM-tree) (O'NEIL *et al.*, 1996) is a disk-based data structure that provides low-cost indexing for a file that has a high rate of record insertions and deletions over an extended period. LSM-tree stores the data in a buffer in memory before writing it to secondary memory. The method of buffering avoids 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. LSM-trees are typically used to store keys and values in the application domain, i.e., to store and retrieve user data. Recently, LSM-trees have been widely adopted in No-SQL databases (O'NEIL *et al.*, 1996; CHANG *et al.*, 2008).

Single-page repair (GRAEFE; KUNO, 2012) handles the recovery of individual pages (after single-page failure) instead of recovering the whole media device. This technique is executed during system performing without aborting the transaction, the access to the affected data is just delayed. Single-page repair uses an index data structure to identify the LSN of the most recent update of each page since the PageLSN field of a failed page cannot be accessed. A B-tree maps each database page to its most recent log record and its most recent backup page. This allows the history of updates from an individual page.

Single-pass restore (SAUER *et al.*, 2015) allows the backup restoration and log replay in a single operation after a media failure. During log archiving process, it applies run generation logic. Thus, the log file is partitioned, and the log records in each partition (run) are

sorted primarily by page ID rather than by LSN. During media restoration, the system merges page images from backups and partitions in the log archive. The process is illustrated in Figure 17. The number of partitions can be reduced by intermediate merges in order to speed up the restoration.

Figure 17 – Illustration of single-pass restore.



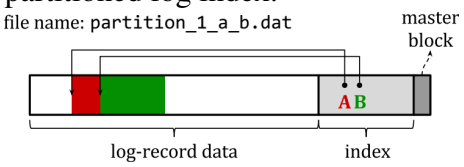
Source: Sauer (2017).

Instant restart (GRAEFE *et al.*, 2015; GRAEFE *et al.*, 2016) is a technique that enables new transaction performing immediately after the analysis phase, and concurrent to redo and undo phases after a system failure. In addition to identifying dirty pages and active transactions at the crash time, the analysis phase identifies transaction locks. Then, the lock manager holds these locks. The analysis also registers the pages that require redo actions. After the analysis, the system can perform new transactions during the redo and undo phases. The main idea of instant restart is to recover the database incrementally and on-demand. Transactions that do not require registered pages or do not conflict with locks acquired in the analysis can run. When a transaction requires a registered page that has not yet been recovered, a single-page repair is performed. A transaction in conflict with the locks acquired in the analysis phase must wait or abort.

Instant Restore (SAUER *et al.*, 2017; SAUER, 2017; SAUER, 2019) technique allows transaction processing during the recovery process after a media failure. This technique was implemented in Zero Storage Manager (SAUER, 2022; SAUER, 2017), which is a fork of Shore-MT (JOHNSON *et al.*, 2009; SHORE-MT, 2021), an experimental test-bed library that can be used as a storage engine. The Instant Restore log file data structure is a partitioned B-tree (GRAEFE, 2003). During normal transaction processing, log records are stored in a B-tree partition at commit time. Only the records of a run generation (e.g. a group commit) are stored

in a partition. Before committing, records are sorted primarily by page ID and secondarily by LSN in each partition. The partitions are implemented by a flat-file index (Figure 18). Each partition is a file whose log records are stored contiguously. Index information is appended in each file as a list of (pageID, offset) pairs. This index allows the offset to the first update record of a page within a partition, given the page ID. The B-tree search key is the page ID that maps all partitions that contains log records that updated a page ID. In Figure 18, the index partition maps update records of pages *A* and *B*.

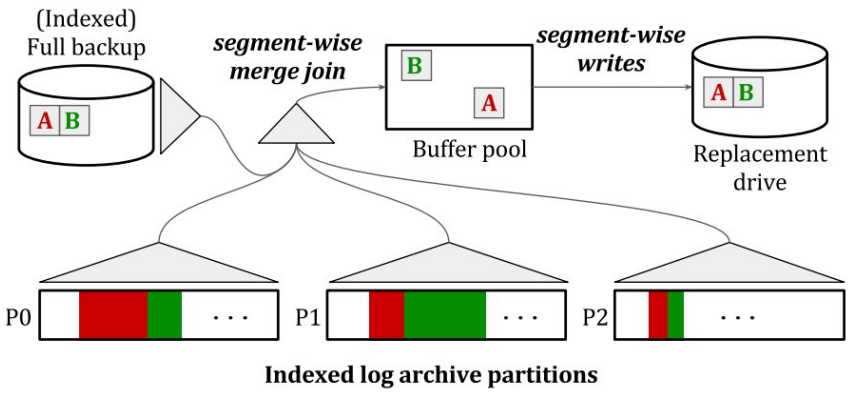
Figure 18 – Flat-file format of the partitioned log index.



Source: Sauer (2017).

After a media failure, the Instant Restore recovery process loads pages incrementally from a backup device. While a set of pages is loaded, the log records of that set are probed in the log archive. After the pages are restored, they are available for new transactions. The approach is also able to recover pages on-demand for new transactions. The Instant Restore technique is illustrated in Figure 19, which shows a segment consisting the two pages (*A* and *B*). During recovery, these two pages are restored by loading them from the backup and fetching them on the indexed log partitions. The partition index may search for multiple partitions to restore a page. Thus, partitions must be merged periodically to provide acceptable read performance during recovery, as the number of partitions increases with the number of log records generated (SAUER *et al.*, 2017; SAUER, 2017; SAUER, 2019).

Figure 19 – Illustration of instant restore.



Source: Sauer (2017).

3.2 MMDB recovery strategies

MMDBs are implemented in a variety of ways, as discussed in section 2.3. MMDBs trend to store only records in memory and avoid page-based indirection by a buffer pool (LARSON; LEVANDOSKI, 2016; FAERBER *et al.*, 2017). Some systems prefer multi-versioning of data instead of in-place updating (MALVIYA *et al.*, 2014; NEUMANN *et al.*, 2015). Indexes usually store direct pointers to records and try to optimize CPU cache efficiency as well as multi-core parallelism (AILAMAKI *et al.*, 1999; FAERBER *et al.*, 2017). MMDBs avoid to implement a lock manager because its indirection in handling the locks is a source of overhead. Instead, they prefer to embed metadata into records to control access to them (GARCIA-MOLINA; SALEM, 1992). Some systems use a multi-version concurrency control variant (CUI *et al.*, 2003; LEE *et al.*, 2013), while others use partitioned serial execution (THOMSON *et al.*, 2012; STONEBRAKER; WEISBERG, 2013).

Recovery strategies should follow the architectural choice of the database system. This section provides examples of recovery techniques used in modern MMDBs. These techniques are a representative sample of the different MMDBs recovery implementations. The recovery techniques presented here are implemented similarly in other types of MMDBs. Table 1 provides a summary comparing the recovery approaches discussed. This table shows important features that influence the recovery mechanism employed for each MMDB: concurrence control (cf. Section 2.3.2), logging (cf. Section 2.5.2), checkpoint (cf. Section 2.5.3), and instant recovery (cf. Section 3.1). For each system, we overview the general features of the MMDB and, then, discuss in-depth the methods of logging, checkpoint, and recovery.

Most MMDBs implements a offline recovery, i.e., the database is only available for new transactions after the full recovery process is completed. These systems prefer to adopt a offline recovery approach because it leads a better performance during transaction processing, as discussed in Section 2.5.4. Moreover, the systems can use some high availability strategy to maintain the database operational in the face of failures. In the recovery approaches covered in this section, only FineLine implements an instant recovery approach (discussed in Section 3.1) that allows the system to perform transactions during recovery. The only way for the other systems to be available during recovery is to adopt some high availability strategy.

Table 1 – Some modern MMDBs and its main recovery features.

MMDB	Concurrence Control	Loging	Checkpoint	Instant Recovery
Hekaton Freedman *et al.* (2014) Diaconu *et al.* (2013) Larson *et al.* (2013)	optimistic MVCC	operation logging	delta and data files	no
VoltDB Malviya *et al.* (2014) Stonebraker e Weisberg (2013)	serial execution in partitions	command logging	snapshot	no
HyPer Funke *et al.* (2014) Kemper e Neumann (2011) Mühe *et al.* (2011)	serial execution in partitions for OLTP, virtual snapshot for OLAP	command logging	snapshot	no
SAP HANA Färber *et al.* (2012) Färber *et al.* (2011) Sikka *et al.* (2012)	MVCC, 2PC	operation logging	snapshot	no
SiloR Zheng *et al.* (2014)	optimistic MVCC	value logging	"fuzzy" snapshot	no
TimesTen TimesTen (2020) Lahiri *et al.* (2013) Team (1999)	MVCC, 2PL	value logging	"fuzzy" checkpoint, snapshot	no
PACMAN Wu *et al.* (2017)	serial execution in partitions	command logging	-	no
Adaptive Logging Yao *et al.* (2016)	serial execution in partitions	command logging, ARIES logging	snapshot	no
FineLine Sauer (2019) Sauer *et al.* (2018) Sauer (2017)	-	physiological logging	-	yes

3.2.1 Hekaton

Hekaton is an in-memory engine of Microsoft SQL Server. A SQL Server database can have both disk-based tables (regular tables) and in-memory tables (Hekaton tables). A regular table can become a Hekaton table simply by declaring the first one as a table memory-optimized. A Hekaton table is entirely stored in the main memory and is durable. The in-memory and regular tables are both handled by Transact-SQL (T-SQL). A transaction can update both in-memory and disk-based tables. However, a stored procedure that accesses only in-memory tables is compiled in the machine code. Hekaton transactions can access any database table. The engine uses an optimistic multi-version concurrency control and lock-free (latch-free) structures to prevent concurrency among threads. Hekaton uses logs and checkpoints to ensure transaction durability and recovery after a failure (DIACONU *et al.*, 2013; FREEDMAN *et al.*, 2014; LARSON *et al.*, 2013).

The log in Hekaton is logical and redo-only. A transaction generates log records only at the commit time since Hekaton does not implement WAL protocol. The log records contain logical information about inserted and deleted versions which are sufficient to restore the database. The system writes to the log the versions created and keys of deleted versions when the transaction commits. The commit processing uses group commit, i.e., it tries to group multiple log records from different transactions performing in the same period into one I/O (DIACONU *et al.*, 2013).

The checkpoint should occur incrementally and periodically during the system execution. The checkpoint is stored in two types of files: *(i)* data file, which stores new versions from insertions and updates; and *(ii)* delta file, which contains information about which of the versions have been deleted. Data and delta files are append-only. A data file is strictly read-only. The checkpoint process appends new versions to data files and IDs of removed versions to delta files (DIACONU *et al.*, 2013).

After a failure, the Hekaton recovery mechanism locates the most recent checkpoint file that tracks all data and delta files. Delta files filter rows of removed versions. Thus, the recovery process can reload only the current versions into memory. The data/delta files pair represents the unit of work for recovery and leads to a recovery process highly parallelizable. Hekaton manages one thread per core to replay the transactions saved in the checkpoint files parallel. When the checkpoint process is completed, the log is replayed from the last checkpoint record (DIACONU *et al.*, 2013).

Data files can degrade recovery performance in proportion to the number of versions deleted considering the fact that the recovery manager must read all data and delta files. Thus, adjacent data files are merged to prevent this inconvenience. Data files are merged in a new data file when their active content drops below a threshold. The new delta file generated from those data files is empty because all deleted versions are dropped after the merge (DIACONU *et al.*, 2013).

3.2.2 VoltDB

VoltDB is an OLTP MMDB designed from H-Store (KALLMAN *et al.*, 2008) (academic version). VoltDB partitions data to allow the storage of databases larger than the memory available of a single node. For high availability and fault-tolerance, it can be replicated across several nodes. Each partition is assigned to one site, and each node can run multiple sites. A transaction in VoltDB is performed serially in a partition, i.e., a transaction runs exclusively on the necessary partitions. Each node has an initiator responsible for sending transactions to the partitions/replicas. A transaction must be a single stored procedure (MALVIYA *et al.*, 2014; STONEBRAKER; WEISBERG, 2013).

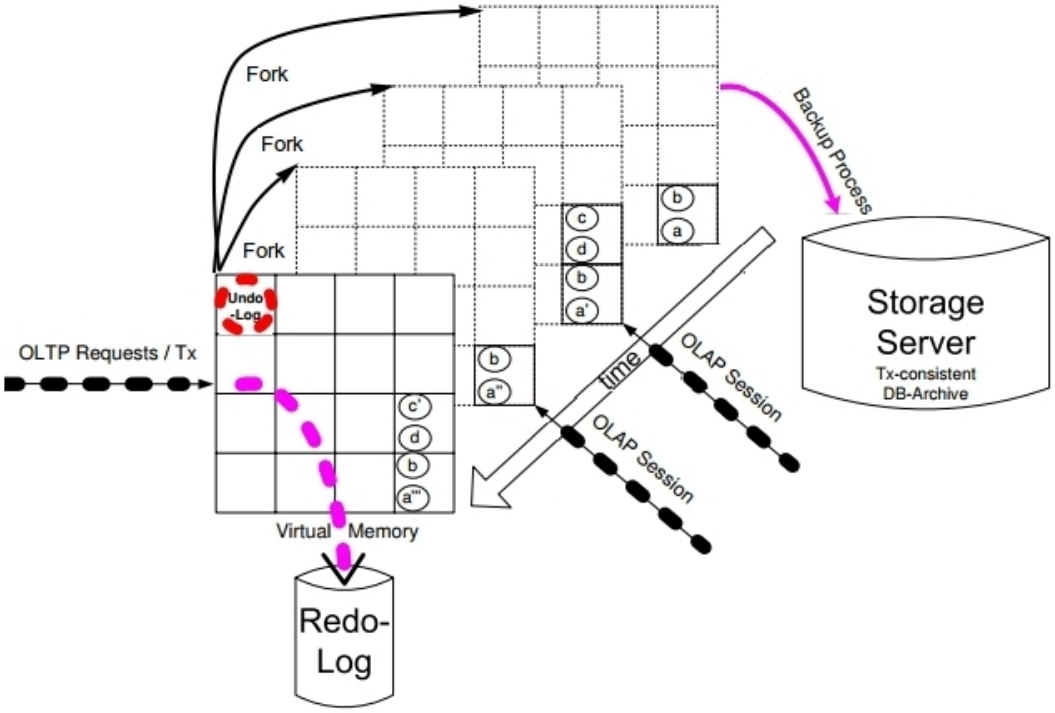
VoltDB implements command logging and asynchronous transaction-consistent checkpoint to recover the database from a failure. Indexes are not flushed to the log file. A single-partition transaction writes records to the log file of its site. A distributed transaction needs multiple partitions and, consequently, it is performed in multiple sites. One of these sites is chosen to coordinate the other sites. Only the coordinator site of the distributed transaction stores log records. Messages exchanged between the coordinator and the other sites are also logged. Replicas of the coordinator also record log records (MALVIYA *et al.*, 2014; STONEBRAKER; WEISBERG, 2013).

When a crash occurs, the recovery manager starts loading the latest database snapshot from secondary storage to main memory. Next, a thread copies the log content of each node into memory. The node's initiator processes the log entries and dispatches the corresponding transaction to the appropriate sites. This recovery schema can work even if the site topology changes. For example, when a site can not restart, the number of execution sites will change. Thus, if a site is removed, the initiator can send the transactions of the removed site to a new site for a given partition-id. Indexes are rebuilt in parallel with snapshot reloading before log replay (MALVIYA *et al.*, 2014; STONEBRAKER; WEISBERG, 2013).

3.2.3 Hybrid OLTP&OLAP High Performance (HyPer)

HyPer is an MMDB that can handle both OLTP and OLAP workloads simultaneously on the same database. In HyPer, OLAP queries can run on the most recent transactional database state. HyPer isolates OLTP and OLAP tasks from each other to accommodate both on the same database. It separates the data between the most immutable data and the most recently updated data. HyPer creates a virtual memory snapshot duplicating the OLTP process to execute a session of OLAP queries. In Unix, for example, the fork() system call can create a child process from a parent process. The OLAP (child process) uses a copy of the address space from the OLTP (parent process), as exemplified on the left of Figure 20. HyPer supports multiple OLAP sessions (one session per core) (FAERBER *et al.*, 2017; FUNKE *et al.*, 2014).

Figure 20 – HyPer's virtual memory schema to support OLTP and OLAP transactions performing in the same database.



Source: Funke *et al.* (2014).

Initially, OLTP and OLAP processes share the same virtual memory. When an object is updated, the copy-on-update schema replicates the modified virtual memory page. Thereafter, after an update, it is created a new version of the data page accessible only by OLTP transactions. The old version is handled only by the OLAP thread. The remaining versions which were not updated after the OLAP session starts are still shared by the OLTP and OLAP processes. Figure 20 shows an example of this schema. The items a, a', and a" represent prior versions of item

a”’ (current version). Only OLTP transactions can access a”’. Each of the pages of the three old versions can be accessed only by their corresponding OLAP session. Each snapshot will be deleted after its session queries are finished. The OLTP transactions are executed serially. However, multiple read-only threads can run in parallel by multiple cores (one thread per core) (FUNKE *et al.*, 2014; KEMPER; NEUMANN, 2011; MÜHE *et al.*, 2011).

The OLTP process must create updated pages in a separate copy (Delta). Periodically and asynchronously, the system merges delta and main stores. This approach allows for updating OLAP sessions with the most up-to-date data. However, it requires an extra reorganization effort in the merge process. Another problem is the storage overhead from OLAP sessions when the number of updated pages increases. Moreover, OLAP queries can make secondary storage accesses to allocate temporary data structures when the main memory is low (FUNKE *et al.*, 2014; KEMPER; NEUMANN, 2011; MÜHE *et al.*, 2011).

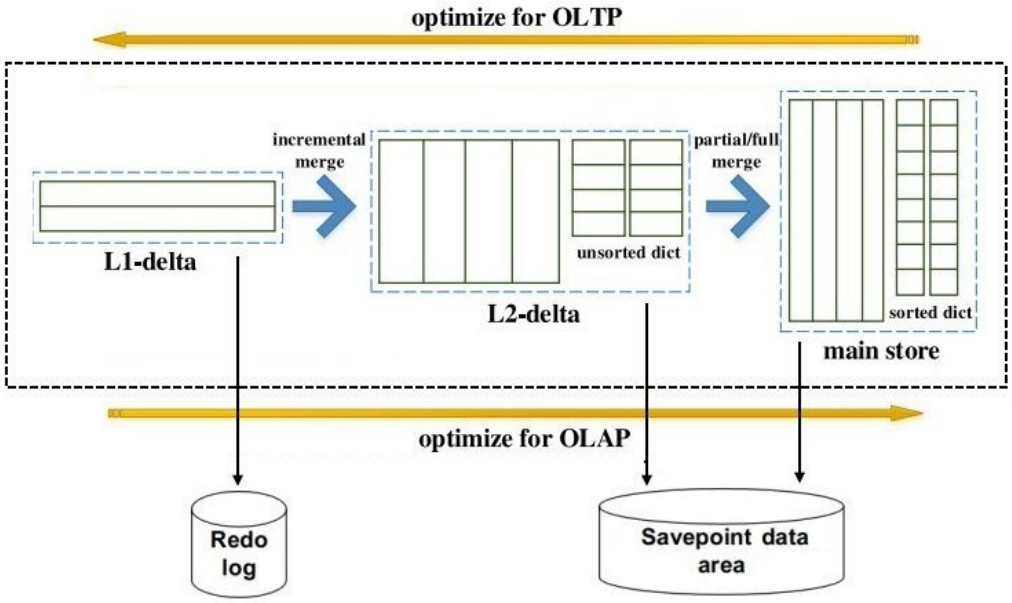
HyPer employs redo-only command logging (on the bottom of Figure 20) to achieve the durability of transactions. HyPer maintains an undo log in volatile memory to undo transactions, if necessary. The undo log records of a transaction can be overwritten as soon as it is committed. The virtual memory snapshots can be used to create database backups on secondary storage (on the right of Figure 20). After a failure, the recovery process starts writing the youngest fully written snapshot archive on the secondary storage into the main memory. Next, the log is replayed forward from the checkpoint record (FUNKE *et al.*, 2014; KEMPER; NEUMANN, 2011; MÜHE *et al.*, 2011).

3.2.4 SAP HANA

SAP HANA is an MMDB that has multiple data processing engines within the same system to provide the classical relational processing (both transactional and analytical), and text processing and graph processing for the management of semi-structured data and unstructured data. SAP HANA can support different query languages. SAP HANA implements a unified table structure (Figure 21) that propagates tuples for three stages of representation within a controlled life cycle management process: optimized storage to write OLTP transactions, and a highly compressed structure to efficiently answer OLAP queries. The common default setup has three representations for tuples in a table: L1-delta, L2-delta, and Main store. L1-delta has the row logical format and is optimized for fast insertion, update, deletion, and record projection. L2-delta is an intermediate structure organized in column-store format and use dictionary encoding

for better memory usage. This dictionary is unsorted and needs secondary index structures for a better access performance. The main store implements a columnar store format in a sorted dictionary that can be highest comprised in several compression approaches (FÄRBER *et al.*, 2012; FÄRBER *et al.*, 2011; FAERBER *et al.*, 2017; SIKKA *et al.*, 2012).

Figure 21 – Unified table hybrid store.



Source: Zhang *et al.* (2015) and Sikka *et al.* (2012).

SAP HANA was implemented for OLAP workloads originally. This kind of workload is very appropriated for range queries, but it makes update operations prohibitively expensive because a large number of entries need to be updated. To avoid this problem, only the delta part handles modification operations (L1-delta for update/insert/delete; or L2-delta for bulk load operations). The system merges delta into the main storage periodically: first L1-to-L2-delta merge; and later an L2-delta-to-main merge. Both merge operations are asynchronously propagated through the storage, do not affect persistent storage directly, and do not interfere with system restart (FÄRBER *et al.*, 2012; SIKKA *et al.*, 2012).

The persistence in SAP HANA is a combination of REDO log and save pointing (snapshot). The UNDO log is not performed. The REDO log stores logical operations from L1-delta and L2-delta transactions. Modifications during the merge process are not logged, but the merge event is written on the log. Changes are persisted periodically in the savepoint of L2-delta and main store. After the savepoint, the REDO log can be truncated. Figure 21 outlines some of the details of SAP HANA persistence. After a crash, the system loads the savepoint and perform REDO log actions (FÄRBER *et al.*, 2012; SIKKA *et al.*, 2012).

3.2.5 SiloR

SiloR (ZHENG *et al.*, 2014) added a concurrent recovery schema in Silo (TU *et al.*, 2013). Silo is a high-performance MMDB which implemented logging but did not consider recovery, log truncation, or checkpoints. Silo implements an optimistic concurrency control to serialize transactions. The system uses an epoch schema in which the transactions read a global number E and embed it in the transaction ID. A designated thread advances E periodically (every 40 ms). Threads use E during the commit procedure to compute the new transaction ID. The epoch approach impacts the way SiloR does logging and recovery and is the key to correct replay. Silo assigns a core for each thread: workers (that carry out client requests), logger, and checkpointers (ZHENG *et al.*, 2014).

SiloR logs transaction output keys and values, rather than operations or store procedures IDs and their parameters. The system stores its log in a collection of files in a single directory. A log file has log buffers concatenated from several threads. The epochs can be store on log out of order since a thread does not need to wait for a prior thread that delays its release. The value logging allows recovery parallelism, but it logs a large amount of data and, consequently, might slow transaction execution. On the other hand, operation logging and command logging techniques require that the recovery manager replays transactions in their original serial order, which is hard to parallel. SiloR uses a separate thread to maintain an epoch file that contains the epoch C whose transactions executed prior are persistent. Thus, the system knows that all transactions in epochs less than or equal to epoch C are durably stored in log (ZHENG *et al.*, 2014).

In SiloR, checkpoint uses multiple threads and multiple media (one thread per media). The checkpoint manager assigns different slices of the database to different checkpointer threads. Each checkpointer writes records from its slice to the secondary storage as long as the transactions commit. However, as concurrent transactions continue to modify the database during the checkpoint process, it is possible that the checkpointers will not see all new modifications. Thus, SiloR performs a "fuzzy" checkpoint, i.e, the checkpoint is not consistent. After a failure, SiloR starts recovery by loading the most recent checkpoint. The checkpoint records a table on n media, and each media has m files per table. Thus, n x m threads are necessary to load the checkpoint in memory. The recovery threads can restore the database in parallel since the checkpoint stores different key ranges (ZHENG *et al.*, 2014).

After the checkpoint is finished, the system recovery starts to process the log. First,

the system identifies the epoch C, i.e., the number of the latest persistent epoch. This is necessary to correctness since group commit could not have finished for those later epochs. If log records after the epoch C are processed, the database can stay in an inconsistent state. In contrast to checkpoint files, log files are not organized. The log files can be processed in any order. At the end of log processing, each key is associated with a value of the last modification from its most recent persistent epoch. Then, the log replays in reverse order to avoid overwriting some value and, consequently, use the CPU more efficiently. This SiloR's log property allows parallelism and prevents unnecessary allocations, copies, and work (ZHENG *et al.*, 2014).

3.2.6 TimesTen

Oracle TimesTen In-Memory Database (TimesTen) is an OLTP in-memory database system, but it can also operate OLAP workloads by synchronizing data into a standby instance. In addition, the storage manager can support columnar compression using dictionary-based encoding to reduce memory usage. TimesTen can be used as a standard in-memory database or a cache of critical performance subsets of an Oracle database. Client applications can read/write the cache tables and the synchronization of data between cache and database is performed automatically. TimesTen can also be scaled-out to a grid of interconnected hosts that work together to provide fast access, fault tolerance, and high availability. TimesTen uses MVCC for read-write concurrency and record-level locking for write-write concurrency (HU *et al.*, 2019; TIMESTEN, 2020; LAHIRI *et al.*, 2013; TEAM, 1999).

TimesTen recovery process starts by loading the most recent checkpoint file and the transaction log. During transaction processing, log records are first written to an in-memory log buffer whose contents are flushed to the log file on secondary memory. The transaction log is used to undo transactions that are rolled back. Transaction log records are written to secondary storage asynchronously or synchronously. In the asynchronous mode, the transactions use pre-commit, i.e., a transaction does not wait for the confirmation of writing log records to durable storage. This technique provides very low response times and very high throughput, but it loses the guarantees of consistency in the event of failure. In the asynchronous mode, the transactions use group commit, i.e., a transaction must wait for the log record to be written to secondary storage. This approach avoids data loss when a failure occurs, but it reduces system performance (TIMESTEN, 2020; LAHIRI *et al.*, 2013; TEAM, 1999).

TimesTen supports two types of checkpoint techniques: non-blocking (or fuzzy)

checkpoint, and blocking checkpoint. Fuzzy checkpoint does not obtain locks and therefore does not pause transaction processing. It is generated in the background automatically. In this approach, transactions may modify the data during the checkpoint process. As a result, the generated checkpoint file can store both committed and uncommitted transactions. Blocking checkpoint creates a transaction-consistent checkpoint file (snapshot) by an application call. This technique acquires an exclusive database lock. Thus, new transactions cannot be executed until the checkpoint process ends, and, consequently, the performance of the system is degraded (TIMESTEN, 2020; LAHIRI *et al.*, 2013; TEAM, 1999).

Data replication in TimesTen uses the transaction log to copy data from a master database to a subscriber database. This approach is also used when TimesTen is an Oracle database cache. In TimesTen grid database, each host maintains its own checkpoint and transaction log files. For that reason, each host is durable and recoverable independently (TIMESTEN, 2020; LAHIRI *et al.*, 2013; TEAM, 1999).

3.2.7 PACMAN

PACMAN (WU *et al.*, 2017) is a recovery mechanism designed for transaction-level logging. It tries to parallelize the recovery and minimizes the runtime overhead for transaction processing. This mechanism was implemented in Peloton (PAVLO *et al.*, 2017; PELOTON, 2019), an MMDB optimized for high-performance transaction processing. PACMAN was designed based on two prerequisites: *(i)* the DBMS must utilize command logging, and *(ii)* the DBMS must replay re-executing transactions to recover the database. PACMAN mechanism parallels the log replay by static and dynamic analysis (WU *et al.*, 2017).

At static analysis, PACMAN identifies opportunities for parallel execution when the stored procedures are compiled. In the first stage, the flow dependency of each stored procedure is identified (the execution ordering between operations) and data dependency (if operations could potentially conflict). Thus, a stored procedure is divided into smaller pieces (set of operations) that are organized in a graph (local dependency graph). The graph has constraints of execution among the pieces and possible paralleling opportunities into a store procedure. Next, the local dependency graphs are integrated into a global graph. This graph represents the order of execution among all stored procedures (WU *et al.*, 2017).

After a failure, PACMAN uses the global graph to generate schedules. Each schedule allows the stored procedure pieces to execute in parallel following the global dependency

constraints. PACMAN dynamically analyzes the schedules for a higher degree of parallelism. First, the mechanism enables intra-batch parallel executions by the availability of the runtime procedure parameter values. Second, it allows inter-batch parallel executions in which different log batches are replayed in parallel by applying a pipelined execution optimization (WU *et al.*, 2017).

3.2.8 Adaptive Logging

Adaptive Logging (YAO *et al.*, 2016) was designed for logging and recovery in distributed MMDBs. It implements a distributed adaptive command logging method on H-Store. This logging method allows tuning the trade-off between transaction processing and recovery by user parameters. Before the recovery process starts, the system scans the log and generates a dependency graph. The dependency graph links transactions that read/write the same record. While replaying the log, transactions without dependency relationship can process concurrently, and the others one must wait until all their dependent transactions finish.

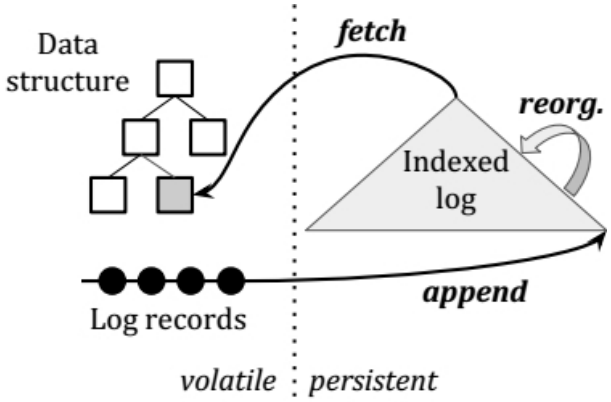
The system employs an online approach to use either command logging or ARIES logging. The dependency between transactions can lead to a few transactions to block the processing of many others. The adaptive logging approach identifies the bottlenecks dynamically using a cost model. A transaction identified as a bottleneck is materialized in the ARIES log. Thus, transactions depending on that bottleneck do not have to wait so long. The cost model uses a budget given by a user. The adaptive logging can spend the same I/Os than ARIES logging if all transactions are identified as bottlenecks (YAO *et al.*, 2016).

3.2.9 FineLine

FineLine (SAUER, 2017; SAUER *et al.*, 2018; SAUER, 2019) is an in-memory system that uses a log-structured data structure as the only way to provide persistence. This approach uses a partitioned B-tree as log file structure and implements the same recovery strategy as Instant Restore (see Section 3.1 for more details), except that the former only uses the log file to recover the data, it does not need a backup device. Thus, after a system failure, the system performs an instant recovery schema, i.e., transactions can be performed during the recovery process. FineLine in-memory data structures are key-value stores. Although the system does not specify any concurrency control schema, the authors suggest that the system can support two-phase locking or optimistic concurrency control approaches.

FineLine persists any data structure that can be organized into a node identified by a unique ID. A node is a single record, which is a form of physical log record. The log is redo-only. Transaction update records are sorted and stored in log partitions at commit time. Undo log records are not persisted, but an undo log is maintained in memory only for transaction abort. The undo log records for a transaction are removed after the transaction is finished. Checkpoints are not implemented. However, partitions are merged periodically in order to reduce the number of partitions and, consequently, to deliver acceptable read performance during recovery. Figure 22 illustrates the FineLine propagation schema. After a system failure, the recovery manager redoes pages incrementally by traversing the indexed log and searching records in partitions. New transactions can run as soon as their necessary pages are restored. When a transaction requires tuples not yet loaded into memory, these pages can be restored on-demand (SAUER, 2017; SAUER *et al.*, 2018; SAUER, 2019).

Figure 22 – FineLine propagation schema.



Source: Sauer *et al.* (2018).

4 INSTANT RECOVERY

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 (MOHAN *et al.*, 1992; GRUENWALD *et al.*, 1996; MALVIYA *et al.*, 2014).

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 (WU *et al.*, 2017; FAERBER *et al.*, 2017; MALVIYA *et al.*, 2014). 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 (SAUER *et al.*, 2017; TAN *et al.*, 2015).

In this sense, this thesis proposes an instant recovery mechanism for MMDBs. This mechanism 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 default recovery strategy implemented by most MMDBs 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 (SAUER *et al.*, 2017; MALVIYA *et al.*, 2014).

The instant recovery mechanism 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 mechanism 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 mechanism in order to show its efficiency and suitability to be implemented in MMDBs. The workload used for the experiments belongs to Memtier benchmark.

The primary purpose of the instant recovery mechanism is to efficiently restore the database incrementally and on-demand, assuring high transaction throughput rates during both the recovery process and normal transaction processing. Downtime after a system failure is expected to be minimal. To achieve this goal, the instant recovery uses an index structure as log file, called indexed log (SAUER *et al.*, 2017; MAGALHÃES *et al.*, 2021; MAGALHÃES, 2021).

The indexed log allows retrieving only the log records to redo a given tuple. Therefore, the restore latency of a transaction depends only on the tuples needed to start processing. Restore latency is defined as the additional delay imposed by the system to retrieve the data necessary for a single transaction to begin executing. Most MMDBs recovery mechanisms need to scan the entire log file to redo a single tuple, i.e., the restore latency for a transaction is the full recovery time (SAUER *et al.*, 2017; MAGALHÃES *et al.*, 2021; MAGALHÃES, 2021).

This chapter details the proposed instant recovery mechanism denoted **Main Memory Database Instant RECovery with Tuple consistent checkpoint (MM-DIRECT)**. First, the anatomy of *MM-DIRECT* is presented and discussed. Second, the data structure used to implement the log file is detailed. Thereafter, the logging technique to provide instant recovery is discussed. Next, the recovery algorithm based on the proposed logging technique is detailed. Finally, the Tuple proposed Consistent Checkpoint technique is described.

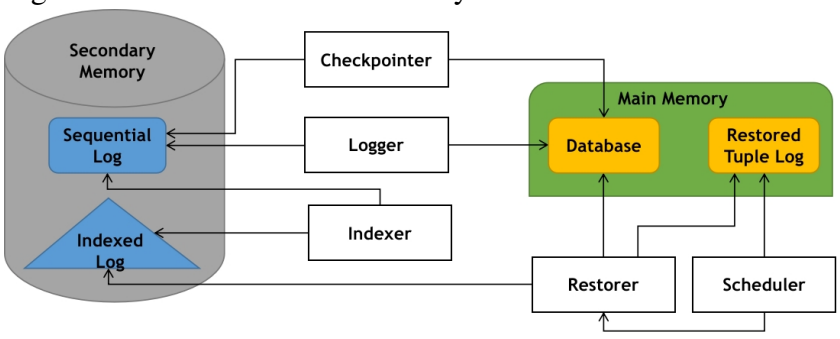
4.1 The anatomy of *MM-DIRECT*

Although modern hardware has offered promising alternatives for MMDBs to reach their full potential, this work does not require many technology improvements. The proposed instant recovery for MMDBs requires a simple system containing a memory hierarchy composed of two levels: a main memory (for the database) and a persistent-memory level (for the log

files). Thus, the proposed mechanism can be deployed in any database system architecture. The mechanism does not require any others technologies, such as NVRAM.

Figure 23 shows the main components of *MM-DIRECT*. The **Logger** flushes transaction write operations into the sequential log file using the write-ahead log approach. The **Indexer** component is responsible for inserting log records from the sequential log file into the indexed log file asynchronous to transaction commitment. Thus, transaction execution is not impacted by the process of inserting records in the indexed log file. In turn, **Restorer** restores tuples, after a failure, by redoing operations from the indexed log file. The **Scheduler** component requests the Restorer to restore tuples required for new transactions in an on-demand way. **Checkpointer** periodically reduces the number of log records in the indexed log. By doing this, *MM-DIRECT* minimizes recovery time.

Figure 23 – MMDB instant recovery architecture.



The following sections describe the components and their interactions in more detail.

4.2 Log files structure

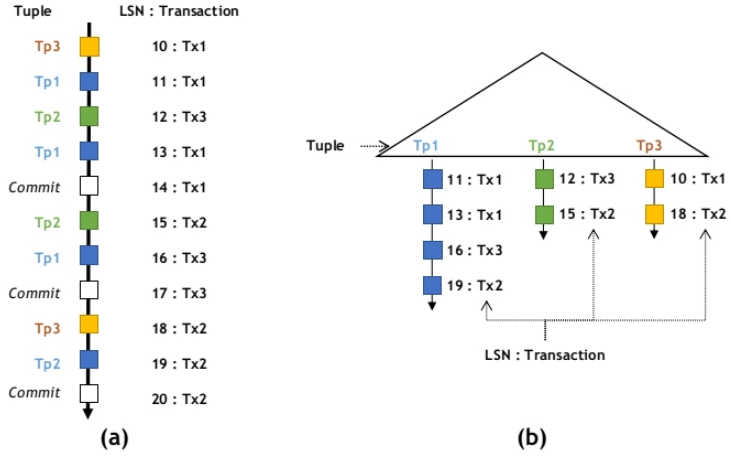
MM-DIRECT works with a sequential log file and an indexed log file. These two log structures are logical (WU *et al.*, 2017; ZHENG *et al.*, 2014), i.e., their records contain higher level operations to describe database updates. The structures are illustrated in Figure 24. During the processing of a transaction, log records are generated for each update action and kept in a local thread, i.e., the transaction performs updates locally in the memory, and other transactions can see its updates only when it commits. During a transaction commit, the log records are flushed to the sequential log file. LSN orders the log records, i.e., the log records are written sequentially in the same order in which the respective update action was performed. This log organization ensures consistent database recovery.

It is a matter of fact that a sequential log does not provide the necessary support

for on-demand recovery. Accordingly, after a failure, new transactions should wait for full database recovery to be executed. In order to overcome such a limitation, *MM-DIRECT* utilizes an indexed log file, which is structured as a B$^+$-tree. The database's tuple IDs are used as search keys. This way, log records for a given tuple T are stored in a list pointed by a single entry of a leaf node. Thus, a leaf node search key value K (a tuple ID) points to a list of log records for a tuple T with $ID = K$. The records are copied to the indexed log in the same format that they were stored in the sequential log. A single index seek operation in the B$^+$-tree can locate all the records required to restore T. LSN is responsible for representing the temporal order of log records belonging to a tuple T. The indexed log loses the global LSN order, but it maintains a local LSN order in each B$^+$-tree leaf node. Log records for a tuple in a leaf node are still sorted by LSN. This log organization ensures the consistent recovery of each tuple. In this way, the B$^+$-tree allows to recovery each tuple individually.

To illustrate how both log files are built, consider Figure 24. In Figure 24 (a), log records generated by transactions Tx1, Tx2 and Tx3 are stored in the sequential log file. On the other hand, Figure 24 (b) illustrates the assemblage of the indexed log file. Log records for transactions Tx1, Tx2 and Tx3 are grouped in the B$^+$-tree leaf nodes by the tuple IDs. For instance, log records with LSN 12 and 15 belonging to Tx3 and Tx2, respectively, contain write operations of write executed on Tp2.

Figure 24 – Sequential log (a), and indexed log (b).



The Fineline recovery mechanism (discussed in Section 3.2.9) supports only partitioned B-tree indexed log. On the other hand, *MM-DIRECT* is implemented to support different index structures, such as B$^+$-tree and Hash table (LITWIN, 1980). B-tree and Hash structures have their trade-offs. A hash table can access a key in $\mathcal{O}(1)$, which is faster access than a

tree index seek operation ($\mathcal{O}(\log_n)$). However, tree algorithms can locate key value ranges in $\mathcal{O}(\log_n)$, whereas hash may degrade to a full table scan ($\mathcal{O}(n)$) in such cases. Besides, hash indexes have constant overheads (e.g. overflow buckets), which is not a factor in theta notation, but it still exists. A tree structure is generally easier to maintain and to scale (OLSON *et al.*, 1999; YADAVA, 2007; LEHMAN; CAREY, 1986). Although these two index structures are possible to perform the instant recovery proposed in this work, the tree proved to be more viable for recovering databases than the hash, as shown in the experiments in the Section 5.6.

4.3 Logging mechanism

Writing records to a sequential log file is potentially faster than doing so to an indexed log file. This is because the former has a sequential pattern, while the latter has a random pattern. As already mentioned, sequential reads from the sequential log do not allow efficient recovery of a given tuple. On the other hand, the indexed log allows recovery on-demand due to its random reads. The primary purpose of the proposed instant recovery is to restore the database efficiently, without degrading transaction throughput provided by the system (SAUER *et al.*, 2017; MAGALHÃES *et al.*, 2021; MAGALHÃES, 2021). Therefore, the logging technique proposed in this work writes log records to the sequential log at commit time and writes these records to the indexed log in the background. In addition, only the indexed log is used to recover the database. We are assuming that both log files are stored on different media (e.g., different disks) for fault tolerance purposes.

During normal transaction processing, transaction update records are appended to the sequential log file at the commitment by the Logger component. The Indexer component is a thread that monitors entries in the sequential log and inserts them in the indexed log. The indexing process is asynchronous to the transaction commit; i.e., a transaction should not wait for the insertion in the indexed log to commit. Such an asynchronous indexing process avoids a negative impact on transaction processing.

The indexed log uses an in-memory buffering mechanism to achieve write rates similar to the sequential log. Writing database log records through a buffer mechanism is inadequate because log records require immediate and atomic persistence during commit processing. However, in *MM-DIRECT*, this requirement is met for each log record through the sequential log before that record is inserted into the indexed log. Therefore, the indexed log can use a buffer method without worrying about possible recovery inconsistency. The indexed log buffering

mechanism is responsible for mitigating the record writing rate difference between the indexed log file and the sequential log file. As a consequence, the log tail increase smoothly. In this work, we refer to log tail as the number of records in the sequential log that have not yet been stored in the indexed log. The experiments presented in Section 5.4 evidence the benefits of the indexed log buffer.

Log records could be removed from the sequential log after they are inserted in the indexed log. Nonetheless, the sequential log is kept to assure that the B^+-tree log can be rebuilt from scratch in an index corruption event. After a failure, in the case of index corruption, a tree rebuild is performed from the last checkpoint. Section 4.6 discusses the checkpoint process and indexed log rebuilding in detail. This process will delay the start of the recovery. In addition, the system can also employ an indexed log replication to avoid downtime in case of log file corruption.

Algorithm 1 shows the pseudocode of the record insertion algorithm in the indexed log. The Indexer thread monitors new entries in the sequential log through a pointer *P*. Pointer *P* represents the LSN of the last record in the sequential log inserted into the indexed log (B^+-tree). As soon as the Indexer notices that there are records after the pointer P, it triggers insertion operations for new log records into the B^+-tree. In other words, new log records appended to the sequential log file are added to the indexed log by a B^+-tree insertion operation (line 4 Algorithm 1). The insertion complexity on B^+-tree is $\mathcal{O}(n \log n)$. Recall that each leaf node key *K* of indexed log B^+-tree points to a list *L* of log records of write operations on a tuple with an ID equal to *K* (see Section 4.2). Therefore, the B^+-tree insertion operation has been extended to append a record to *L*. It is important to emphasize that the code between lines 5 and 9 in algorithm 1 handles the log records responsible for generating checkpoints. The checkpoint technique will be discussed in Sections 4.6 and 4.7.

To illustrate how the proposed logging mechanism works, Figure 25 represents Figure 24 after inserting new log records. In this example, the transaction Tx4 generates new update log records whose LSNs are 21, 22 and 23. As soon as these records are inserted in the sequential log file, the Indexer component identifies these new update records and copies them to the indexed log. It is important to highlight that the records were stored in the leaf nodes of the B^+-tree that map the tuples that these records have modified. Besides, the records were stored in the same order as they were entered in the sequential log in order to preserve the recovery consistency.

Algoritmo 1: Record insertion in the indexed log

```
1:  procedure LogRecordInsertion(lastRecordIndexed)
2:      for each record forward from lastRecordIndexed in log do
3:          if record.type is an update record then
4:              treeInsertion(record)
5:          else if record.type is a tuple checkpoint record then
6:              ID ← record.tupleID
7:              treeDeletion(ID)
8:              treeInsertion(record)
9:          end if
10:     end for
11: end procedure
```

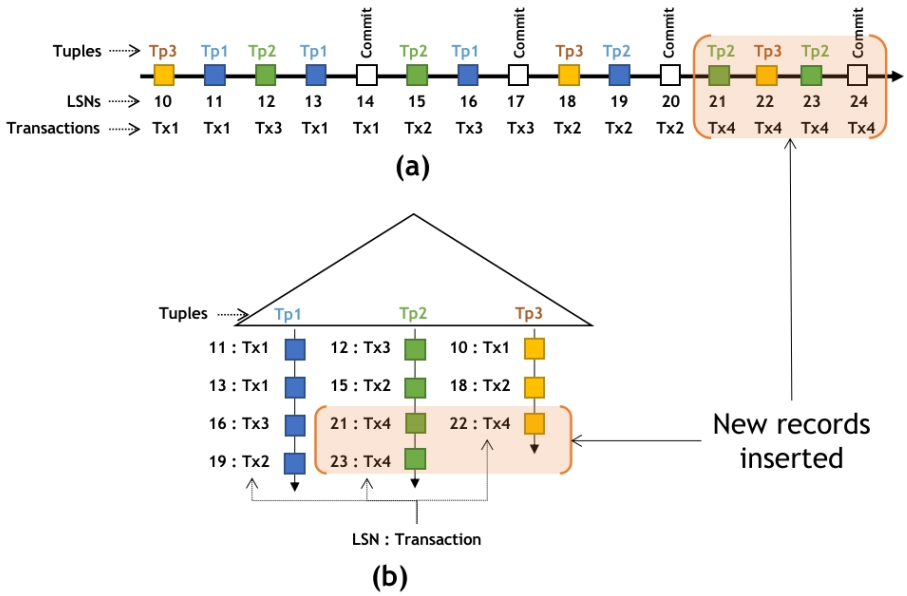


Figure 25 – Sequential log (a), and indexed log (b) after record insertion.

4.4 Recovery procedure

Whenever a system failure occurs, the recovery manager is initialized at system restart. Initially, the recovery manager checks if all records in the sequential log have been inserted into the indexed log file. This check is necessary because some records might not have been inserted into the B^+-tree before a failure. This is because the indexed log file insertion process runs asynchronously to the sequential log file insertion process. Thus, before starting the recovery process, the Indexer component must insert those records into the indexed log file to ensure recovery consistency. Due to the buffering mechanism implemented in the indexed log (see Section 4.3), the log tail does not grow very large. Thus, the number of records in the sequential log to be inserted into the indexed log before starting the recovery should not be too

large.

Immediately after the aforementioned recovery action, the Restorer component triggers incremental recovery by traversing the B^+-tree. Each visit to an entry in the B^+-tree leaf node retrieves the log records to redo a tuple completely. As soon as a tuple is restored into memory, it is set as restored in the Restored Tuple Log component. Restored Tuple Log is an in-memory data structure (see Figure 23). Such an in-memory structure does not consume large main-memory space. Each database tuple needs a pair of variables to identify whether or not the tuple has been loaded: a 4-byte variable to store the tuple id, and a 1-byte variable to store a boolean variable. In this way, for instance, only 5 megabytes are needed to map 1 million database tuples. After visiting all B^+-tree leaf nodes, all database tuples are restored, and the recovery process finishes.

Algorithm 2 describes the incremental instant recovery process. The algorithm starts traversing the indexed log. Recall that the B^+-tree stores data pointers at the leaf nodes. Moreover, the leaf nodes are structured as a linked list; i.e., each node points to its linked neighbor. Thus, B^+-tree traversal can be done in $\mathcal{O}(n)$. For each search key K in a given leaf node, the system checks if an entry in Restored Tuple Log (Figure 23) for a tuple T with ID equal to K exists. In case T's ID is not in the Restored Tuple Log, all log records stored associated with K are redone and an entry for T's ID is included in the Restored Tuple Log.

Algoritmo 2: Incremental instant recovery

```
1:  procedure IncrementalRecovery
2:      for each leafNode in Tree do
3:          for each key in leafNode do
4:              if tuple[key.value] is not restored then
5:                  recordsList ← key.redcordsList
6:                  for each record in redcordsList do
7:                      reapply record
8:                  end for
9:                  set tuple[key.value] as restored
10:             end if
11:         end for
12:     end for
13: end procedure
```

Recovery by an indexed log supports availability naturally since tuples can be accessed by transactions immediately after being restored to memory. However, transactions can request tuples that have not yet been restored to memory. Thus, in parallel to the recovery process,

the Scheduler component schedules read and write operations. Such an ability is supported by the following process. As soon as the Scheduler receives an operation on a tuple T, it reads the Restored Tuple Log to identify if T has been already restored. If the answer is affirmative, the database operation on T is scheduled. Otherwise, the Scheduler requests the Restorer to redo T on-demand. This can be done by an index seek operation to look up for a search key value in the indexed log file equal to T's ID.

Algorithm 3 formalizes the on-demand recovery action. When the Scheduler thread identifies the need for on-demand recovery of a tuple, this thread sends the tuple key value to the Restorer thread. Restorer then performs a B^+-tree search using that key value. If the search operation returns a key, the system redoes the log records stored in the returned key. After the tuple is restored to memory, it is set as restored. Even if a tuple is not found in the indexed log, it is set as restored. This prevents accessing the tree in secondary memory on the next request to the tuple, since it is known that the tuple is not in the tree.

Algoritmo 3: On-demand instant recovery

```
1:  procedure OndemandRecovery(keyValue)
2:      key ← treeSearch(keyValue)
3:      if key is not null then
4:          recordsList ← key.redcordsList
5:          for each record in redcordsList do
6:              reapply record
7:          end for
8:      end if
9:      set tuple[keyValue] as restored
10: end procedure
```

This recovery mechanism is idempotent since the indexed log does not lose the LSN order of each tuple in each leaf node of the B^+-tree, even in the face of successive failures. This LSN order on each B^+-tree leaf node ensures consistent recovery. During the recovery process, the Scheduler thread monitors all requests to the data. The Scheduler ensures that any access to a tuple is made only after it has been completely restored to memory. Thus, a tuple's update records are flushed to the sequential log only after the tuple is restored. Besides, only the Indexer thread is responsible to copy records from the sequential log to the indexed log. Both the Scheduler and the Indexer use only one thread to avoid simultaneous access to data and, consequently, a possible inconsistency in database recovery. Thus, as tuple access and tuple insertion in the indexed log are centralized by the Scheduler and Indexer, respectively, the LSN

order is maintained in each leaf node of the B^+-tree, even if successive failures occur.

4.5 Recovery consistency

In *MM-DIRECT*, as previously mentioned, the Restored Tuple Log prevents the Restorer from loading a previously restored tuple into memory. However, there may still be an inconsistency in the database recovery due to the fact that the Restorer and Scheduler components access the Restored Tuple Log simultaneously. For example, the Scheduler can identify the need to restore a tuple T while the Restorer is currently recovering t. Thus, the Restorer may load tuple t into memory more than once. As a consequence, the second restoration of t may overwrite changes on t executed after the first restoration and before the second one.

To illustrate the aforementioned problem, consider a bank account database recovery scenario. Suppose the system is recovering an account C. In the meanwhile, the Restorer receives an on-demand recovery request from a transaction T_1 with a deposit operation in C. As soon as C is recovered for the first time by the Restorer, it is visible to new transactions. While the Restorer is recovering C to attend T_1's request, another transaction T_2 executes a write operation on C and commits. In such a scenario, changes executed by T_2 on C might be overwritten by the Restorer.

Thus, additional measures are needed to avoid recovery inconsistency. One option would be to hold locks to protect reads and updates in the Restored Tuple Log. However, handling locks should be avoided in MMDBs, as discussed in Session 2.3.2.

MM-DIRECT ensures recovery consistency through an atomic operation to load the data into memory. This operation is a set of steps performed atomically; i.e., the steps of different operations are not performed concurrently. First, the operation checks whether the tuple is in memory. Then, the tuple is loaded into memory only if it is not already in memory. Therefore, the Restorer component does not have to worry about restoring a tuple more than once. These steps are described in Algorithm 4.

Algoritmo 4: Tuple loading

```
1: procedure LOADTUPLEATOMICALLY(key, value)
2:     if key is in memory then
3:         loadTuple(key, value)
4:     end if
5: end procedure
```

4.6 Tuple consistent checkpoint

The indexed log is a growing file. The more records stored in the log the more records to redo during database recovery, consequently increasing database recovery time.

MM-DIRECT implements a novel checkpoint strategy - the Tuple Consistent Checkpoint (TuCC) - to overcome such a negative side-effect. The TuCC should be performed periodically in order to reduce the number of records in the indexed log file. Thus, the recovery process is significantly accelerated. Although this technique directly handles the sequential log, as displayed in Figure 23, its goal is to reduce the number of records in the indexed log file.

Algorithm 5 formalizes the checkpoint process implemented by *MM-DIRECT*. It is possible to observe that the TuCC generation process scans the entire database at system runtime to obtain all the database tuples. For each tuple, the TuCC generates a new log record using the tuple contents and then flushes the record into the sequential log. Such a record is similar to the one generated by an update operation. Nonetheless, a log record generated by the checkpoint is not identified as a normal update record. It is marked with a different log record type attribute.

Algoritmo 5: TuCC generation

```
1: procedure CHECKPOINT
2:     for each tuple in database do
3:         create tupleCheckpointRecord using tuple
4:         flush tupleCheckpointRecord to log
5:     end for
6: end procedure
```

The tuple checkpoint record reflects the state of the tuple at the time its log record was flushed. In other words, records inserted into the log before a tuple checkpoint record are no longer needed to redo that tuple. Therefore, whenever the Indexer finds a tuple checkpoint record R for a tuple T, all records on B^+-tree node N related to T are removed before storing R in N.

Algorithm 1 - between lines 5 and 9 - shows how *MM-DIRECT* inserts a tuple checkpoint record in the indexed log. Observe that if the Indexer finds a tuple checkpoint record, the tuple ID value stored in the record is retrieved. This ID value is used in a B^+-tree key delete operation that removes all log records related to this ID. Then the tuple checkpoint record is added to B^+-tree in an insert operation.

The TuCC is consistent because all log records before a tuple checkpoint record for a given tuple can be discarded. Thus, this technique leads to faster database recovery. In addition,

in contrast to other checkpoint techniques (such as Fuzzy and Snapshot), the TuCC does not lose any changes made if the checkpoint process is interrupted. This is possible since this technique handles each leaf node individually in the tree, i.e., the checkpoint of one tuple does not interfere with the others. During the checkpoint process, each checkpoint in a tuple only reflects the state of that tuple. Only the full checkpoint reflects the state of the entire database, as all database tuples have been flushed.

4.7 Checkpointing most frequently used tuples

In Algorithm 5, the entire database should be read to generate a checkpoint, even non-updated tuples. Such an technique may cause an unnecessary effort to perform a checkpoint in several non-updated or low-frequency updated tuples. Thus, *MM-DIRECT* implements another checkpoint technique, which is applied for the most frequently used (MFU) tuples, denoted Tuple Consistent Checkpoint for MFU tuples (TuCC-MFU).

Algorithm 6 describes the TuCC-MFU technique. During normal transaction processing, the system stores tuple access information by a simple method: it uses an access counter for each tuple. Therefore, the tuple access information is a set S of tuples (*tupleID*, *counter*), where *tupleID* is the tuple ID, and the *counter* contains the number of accesses to that tuple. The set S should not be a source of memory overhead, as it should not need a lot of memory space. Each tuple of S needs two long integer variables. Each variable needs 4 bytes. Thus, for instance, only 8 megabytes are needed to map 1 million database tuples. In addition, after each checkpoint, S can be cleared.

TuCC-MFU uses tuple access information (set S) to identify MFU tuples. Before starting a checkpoint process, the system sorts the set S in descending order by the counter value. Then, the system creates a set of MFU tuples by the first k pairs in the set S. The value of k is given by the database administrator (DBA). Finally, the system creates and flushes to the sequential log a tuple checkpoint record for each MFU tuple. The Indexer inserts those records into the indexed log in a similar way as discussed in Section 4.6. It is important to note that the MFU tuples checkpoint can not be used as the starting point for rebuilding a corrupted indexed log file since it does not reflect the state of all database tuples.

The TuCC-MFU process is potentially faster than TuCC. This is because the former yields fewer log records than the latter. Nonetheless, after a failure, in an indexed log file corruption event, the system can rebuild the B^+-tree from the sequential log, as discussed in

Algoritmo 6: TuCC generation for MFU tuples

```
1:  procedure CheckpointMFU(tupleAccessInformation, k)
2:      taiSorted ← sort(tupleAccessesInformation)
3:      mfu ← ∅
4:      i ← 0
5:      for each tupleID in taiSorted do
6:          tuple ← database[tupleID]
7:          add tuple to mfu
8:          i ← i + 1
9:          if i == k then
10:             break
11:         end if
12:     end for
13:     for each tuple in mfu do
14:         create tupleCheckpointRecord using tuple
15:         flush tupleCheckpointRecord to log
16:     end for
17: end procedure
```

Section 4.3. Since the TuCC process reflects the last database state, as discussed in Section 4.6, the indexed log file reconstruction can start from the log record generated by the last TuCC process. The TuCC-MFU technique does not support such an indexed log rebuilding.

4.8 Qualitative comparative analysis

This section does a qualitative comparative analysis between *MM-DIRECT* and the related works discussed in Section 3. The related works need to recover the database completely so that new transactions can be executed, except Fineline which uses an instant recovery mechanism. For this reason, we will only compare Fineline with *MM-DIRECT*.

MM-DIRECT implements a logical logging technique in which logical records are flushed to a sequential log file at commitment. Besides, it implements an indexed log, through a B^+-tree, in which record writes are asynchronous to commit time, i.e., updates to the indexed log does not interfere in the transaction processing. In contrast to *MM-DIRECT*, Fineline (discussed in Section 3.2.9) only implements a log file through a partitioned B-tree (indexed log). In Fineline, physical log records are flushed to the indexed log at commit time, i.e., a transaction must wait for updates to the indexed log to commit its writes. As discussed in Section 2.1.2, logical logging tends to be faster than physical logging during transaction processing. Moreover, as discussed in Section 4.3, record writes to a sequential log file is potentially faster than doing

so to an indexed log file. Thus, the *MM-DIRECT* logging mechanism is much more lightweight than the Fineline logging technique.

MM-DIRECT is enable to retrieve all log records to restore a given tuple by only one search in the indexed log B^+-tree. A B^+-tree leaf node points to a list that contains only log records to restore a tuple completely. In Fineline, a indexed log B^+-tree node points to all partitions that contains the log records that updated a given page. Each partition can contains update records of several pages. Thus, Fineline must traverse multiple partitions to restore a page, i.e., it must access multiple files (flat-files). Besides, in each partition, the log records to restore that page must be probed by a flat-file index. Thus, accessing log records in *MM-DIRECT* is much simpler and potentially much faster than in Fineline.

The *MM-DIRECT* checkpoint technique reduces the number of log records in the indexed log in order to accelerate the recovery process. On the other hand, Fineline implements only a technique to merge partitions to provide acceptable read performance during recovery. However, recovery time tends to increase as the number of log records increases. Thus, *MM-DIRECT* checkpoint technique is effectively enable to reduce the recovery time, while the Fineline does not implement checkpoints. A checkpoint in Fineline would be a very expensive process, as the system would have to update multiple partitions for each page.

5 EVALUATION

In order to assess the potentials of the proposed mechanism for main memory database system recovery, simulations over Memtier Benchmark (MEMTIER BENCHMARK, 2020) have been conducted, and the main results achieved so far are presented and discussed in this section. In what follows, we first provide information on how the simulation prototype was set up. Then, the empirical results are quantitatively presented and qualitatively discussed.

5.1 Evaluation setup

All experiments present in this thesis were executed with 4 worker threads on Intel Core i7-9700k CPU 3.60GHz x 8, with 64GB RAM and 400GB SSD. The operating system was Ubuntu Linux 18.04.2 LTS. Table 2 summarizes the system settings used in the experiments.

Table 2 – System components/settings used for the experiments.

Processor	Intel Core i7-9700k
CPU	3.60GHz x 8
Main memory	64GB RAM
Secondary Sotrage	400GB SSD
Operating system	Ubuntu Linux 18.04.2 LTS
Worker threads	4

Memtier Benchmark (MEMTIER BENCHMARK, 2020) was employed to simulate workloads and the main memory database. Memtier is a high-throughput benchmarking tool for Redis and it was developed by Redis Labs (REDIS LABS, 2020b). 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 has many other configuration options to simulate custom workloads (OUAKNINE *et al.*, 2017; MEMTIER BENCHMARK, 2020; MAGALHÃES *et al.*, 2021).

All experiments performed in this thesis used the same workload which in turn used 4 worker threads, with each thread driving 50 clients. Each client made $500,000$ requests. The requests randomly accessed 20% of the database in the 5:5 ratio between read and write operations. Furthermore, the experiments were performed on a database generated through 20

runs of the workload described above. The resulting database contains $5x10^5$ tuples, with a 62.7GB sequential log file, which correspond to $1.8x10^9$ log records. Additionally, an indexed log was generated along with this sequential log by using the recovery mechanism proposed in this work.

In order to verify our hypotheses defined in Section 1.2.1, the experiments were carried out in three following scenarios:

1. Standard recovery using sequential log (SRSL);
2. Instant recovery using asynchronous indexed log record insertion (IRAIL);
3. Instant recovery using synchronous indexed log record insertion (IRSIL).

The SRSL scenario reproduces the traditional MMDB recovery process discussed in Section 2.5. In this scenario, during the transaction processing, transaction update records are written to a sequential log file at the transaction commit. The recovery manager recovers the database by scanning the entire log file to reapply record actions. Transactions can be scheduled only after the recovery has been completed.

The IRAIL scenario represents the MMDB instant recovery mechanism proposed in this work. IRAIL uses a sequential log file and an indexed log file. In IRAIL, transaction update records are written in the sequential log at transaction commit time. Records are written from the sequential log to the indexed log asynchronously. Thus, a transaction does not need to wait for the indexed log record insertion. After a system failure, the Restorer traverses the indexed log (B^+-tree) to recover the database. Transactions can be scheduled during the recovery process execution.

IRSIL is a scenario derived from IRAIL. The IRSIL scenario has been created to measure the indexed log record insertion overhead during transaction processing. In IRSIL, transaction update records are only written to the indexed log at transaction commit, i.e., a transaction waits for the insertion of the records into the indexed log to finalize the commit operation. Recovery in IRSIL is similar to the recovery process in IRAIL.

A prototype has been developed to evaluate the feasibility of the recovery mechanism proposed in this work. The evaluation prototype has been implemented in Redis (REmote DIctionary Server) 5.0.7 (REDIS, 2020; REDIS LABS, 2020b)[1]. Redis is an open-source in-memory key-value data store that can be used as a database, cache, and message broker. Redis achieves very high IOPS delivering sub-millisecond response times for real-time applications,

[1] The prototype can be downloaded at: https://drive.google.com/drive/folders/1LTbtY36O0kWIpxZBM-hc1BPvIjICuy2F?usp=sharing

such as Gaming, Ad-Tech, Financial Services, Healthcare, and IoT. Redis has a wide variety of data structures: strings, lists, sets, sorted sets, hashes, and bitmaps. These structures have some shared commands (SET, INCR, DEL, TYPE, RENAME, and others). Redis is written in ANSI C and works in most Portable Operating System Interface (POSIX) systems like Linux.

Redis ensures database persistence by means of a recovery strategy similar to the one implemented by modern MMDBs (see Section 2.5). The logging technique is logical and implements a sequential log file, denoted append-only file (AOF). Periodically, database snapshots are flushed to the secondary storage as a binary dump, called Redis Database Backup (RDB). Besides, Redis can rewrite the AOF in the background when it gets too big. The implemented prototype utilizes Redis AOF to represent the sequential log file required by *MM-DIRECT*. However, RDB and AOF rewrite Redis operations have been disabled in order to capture the scenario of the recovery process on very large log files. To build the indexed log file *MM-DIRECT* utilizes Berkeley DB 4.8 B^+-tree library (OLSON *et al.*, 1999; YADAVA, 2007; BERKELEY DB DOCUMENTATION, 2020).

Finally, the experiments have been classified into six groups: recovery, checkpoint, log files' write bandwidth, different index structures, system overhead, and scalability experiments.

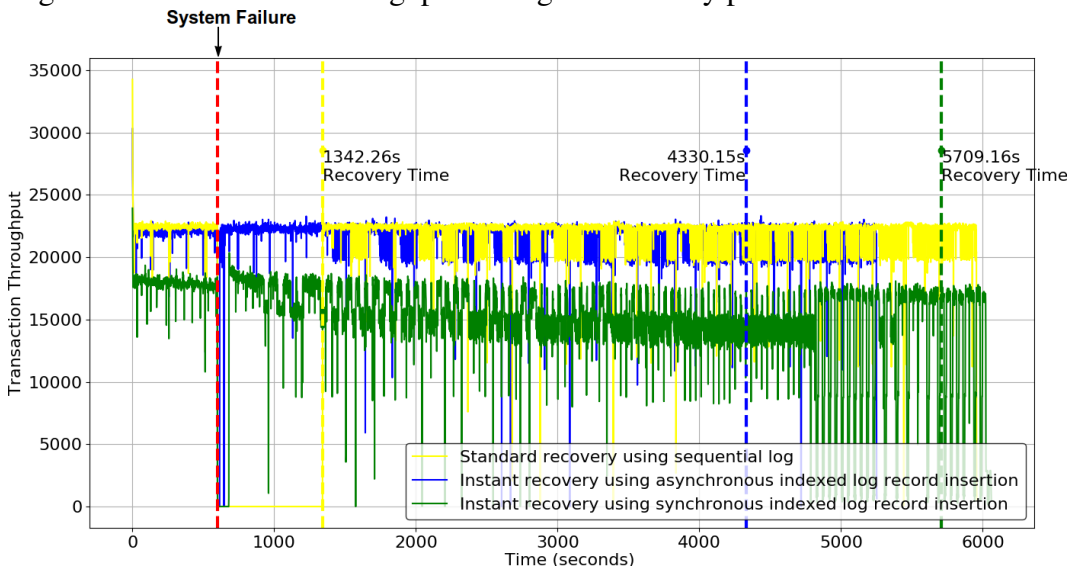
5.2 Recovery experiments

The key goal of the first group of experiments is to compare the proposed instant recovery mechanism to the traditional MMDB recovery. Thus, transaction throughput during the recovery process, database recovery time, and logging overhead were measured. In this sense, for each metric, a workload is submitted to the prototype. After 10 minutes (600 seconds) the database system (Redis) is shut down, simulating a system failure. At database restart, *MM-DIRECT* is triggered, and the workload is submitted again. Figure 26 depicts the results of recovery experiments. The vertical dashed red line indicates the crash time. The other vertical dashed lines indicate the final recovery time of the three scenarios described in Section 5.1: SRSL (yellow line), IRAIL (blue line), and IRSIL (green line).

In Figure 26, one may observe that *MM-DIRECT* (IRAIL scenario) has executed the submitted workload faster than the conventional main memory recovery mechanism (SRSL scenario). Such a result shows that *MM-DIRECT* provides high data availability during the database recovery procedure which is a consequence of the use of the proposed instant recovery

mechanism, supported by the use of an indexed log file. On the other hand, although in SRSL the database was restored before the IRAIL scenario, SRSL presented long downtime after failures. This result was already expected because recovery in SRSL deals with a sequential log, which is potentially faster than manipulating an indexed log (in IRAIL). Sequential access pattern in sequential files can read records faster than the random pattern in indexed log files.

Figure 26 – Transaction throughput during the recovery process.



The experiments also reveal that the *MM-DIRECT* mechanism does not overload transaction throughput during normal databases system function (in the absence of system failures). Observe that in Figure 26 transaction throughput in IRAIL scenario is similar to the one provided by MMDB standard recovery mechanism (SRSL scenario). This result was expected as well because *MM-DIRECT* and standard recovery strategy flush log records to secondary memory in a similar manner. One may claim that *MM-DIRECT* needs to insert additional records into the indexed log, which could negatively impact MMDB throughput. However, record insertion into an indexed log file occurs asynchronously to the transaction commit. Observe that in IRSIL scenario, insertion into indexed log file is executed synchronously to transaction commit according to write-ahead-log approach. In such a case, a transaction must wait for indexed log insertion to commit.

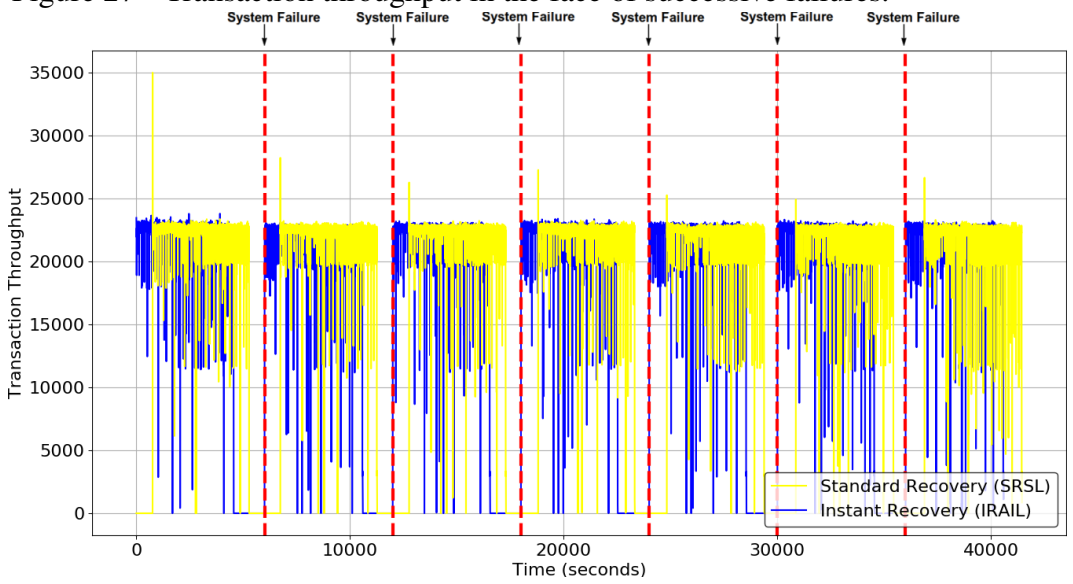
There are additional data on the experiments, which are not depicted in Figure 26. *MM-DIRECT* recovered 411,750 tuples incrementally and 88,250 tuples on demand. Tuples restored in an on-demand way represent 88% of data accessed by the workload during the database recovery. Before starting the recovery process, *MM-DIRECT* presented a very short downtime of 0.0079 seconds, spent to insert 3,022 records into the indexed log file. These

records were not inserted before the last crash because inserts to the indexed log are performed asynchronously to the transaction commit. There were few records to be inserted in the indexed log because the log tail does not grow much due to the indexed log buffer mechanism, as discussed in Sections 4.3 and 4.4. In addition, Berkeley DB itself uses a recovery mechanism internally whose runtime has been added to MM-DIRECT downtime. However, Berkeley DB didn't spend a lot of time during its recovery because it probably handled few records.

The results presented in Figure 26 confirm our hypotheses: (H1) an indexed log is quite efficient to support MMDB instant recovery, and (H2) an asynchronous insertion of log records into indexed log file avoids the decreasing of transaction throughput.

MM-DIRECT has been further evaluated in a scenario with successive failures. Figure 27 brings the results of experiments conducted to compare the behavior of *MM-DIRECT* (IRAIL scenario) and standard recovery (SRSL scenario) in the context of successive failures. For each scenario, a workload was submitted at the database startup, and the system was shut down six times to simulate successive failures. To stress *MM-DIRECT* recovery mechanism, each shutdown has been induced after 6,000 seconds the database system has been started. At each database restart, the workload was submitted again. That shutdown time was chosen to ensure that each workload ran completely before each failure. Looking more closely at Figure 27, one can observe that the results are similar to those observed in Figure 26. *MM-DIRECT* was the fastest mechanism to finish each workload execution, maintaining high transaction throughput rates. Furthermore, *MM-DIRECT* did not suffer from downtime after each failure. It is important to clarify that downtime before each failure is due to the end of workload execution.

Figure 27 – Transaction throughput in the face of successive failures.



Transaction latency is the time delay between the request and the effect of the transaction change in the database system. As already mentioned, *MM-DIRECT* implements on-demand recovery to support the execution of new transactions during the database restore. For that reason, the latency of a transaction that requests on-demand recovery can be increased by an additional time Δ, denoted *restore latency*. Recall that during on-demand recovery, the Scheduler requests the Restorer to redo a given tuple T, which demands an index seek operation for looking up a search key value in the indexed log file equal to T's ID. Thus, restore latency only affects transactions with on-demand recovery requests.

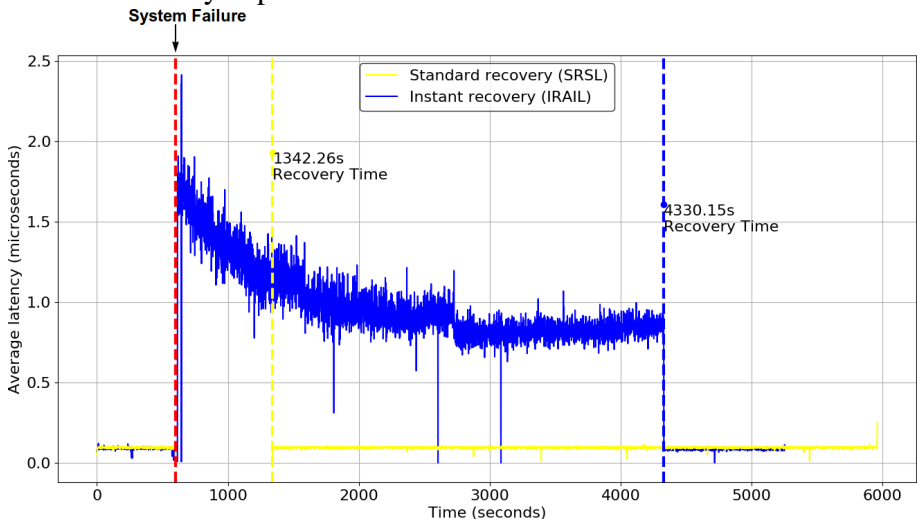
The next experiments investigate the impact of restore latency during the database recovery. We have used the same experiments performed in Figure 26, but we measured transaction latency instead of transaction throughput rate. We have observed recovery latency in *MM-DIRECT* (IRAIL scenario) and default recovery (SRSL scenario). The results are shown in Figures 28, 29, and 30.

Figure 28 depicts the time series of average latency of committed transactions before, during and after the recovery process. Analyzing Figure 28, one may notice that before the failure, the average transaction latency is quite similar (close to zero) for both scenarios. After ten minutes of normal transaction processing, the database system was shut down to simulate a system failure. The immediate effect is that the average transaction latency spikes up in the scenario implementing *MM-DIRECT* (blue lines). The average transaction latency before failure and after database recovery were 0.0898 and 0.0867 microseconds, respectively. The average transaction latency during the recovery was 0.9908 microseconds. The highest average latency was 2.4151 microseconds.

Observe that the highest average transaction latencies are concentrated at the beginning of the recovery process. This delay occurs because data used by those transactions had to be recovered on-demand, i.e., it was necessary to access the indexed log in secondary memory. Once those data have been restored, subsequent accesses to them occur directly on the database located in the main memory with lower latency than secondary memory. Therefore, average latency gradually decreases until database recovery completes, when average latency becomes similar to before the failure.

There is no transaction latency measurement during recovery in the standard recovery scenario (yellow lines) because there is no transaction processing. Recovery time can be considered as part of the latency time of transactions executed after the failure, as these

Figure 28 – Average transaction latency during instant recovery and default recovery experiments.



transactions must also wait for the database downtime to start executing. However, this downtime was not added to the transaction latency values in the graph so as not to make it difficult to see the measured latency values. In this experiment, the database recovery time (1342.26 seconds) corresponded to a very long wait time for transactions to start executing.

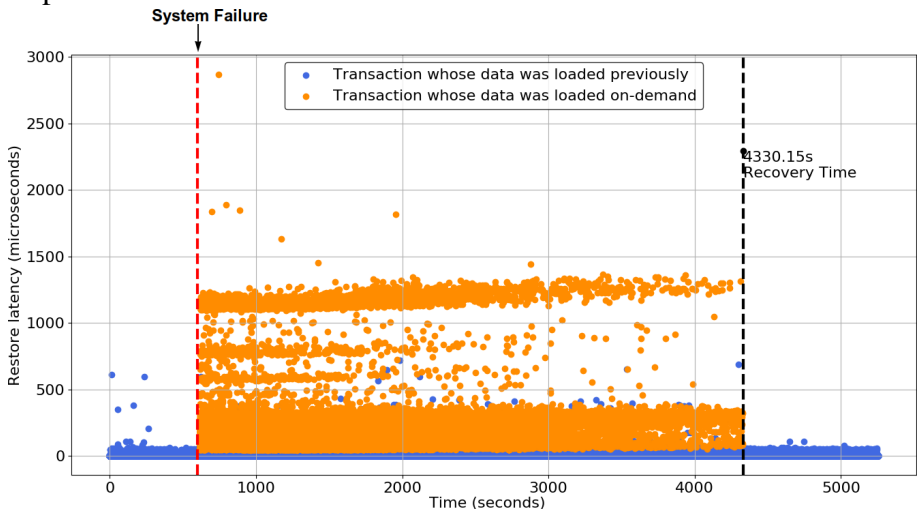
The average transaction latency throughout the default recovery scenario was 0.0957 microseconds. Besides, the average transaction latency before failure and after database recovery were 0.0952 and 0.0958 microseconds, respectively.

Figure 29 is a scatterplot graph of each transaction latency value measured during the instant recovery scenario experiment. Most of the orange dots have higher latency values than the blue dots. Such behavior occurs because the transactions represented by orange dots have to restore latency, i.e., they request on-demand recovery. Those latencies had maximum and minimum values of 2,869 and 43 microseconds, respectively. Transactions represented by the blue dots did not require restore latency, i.e., they access data directly from the memory. Those latencies had maximum and minimum values of 717 and 0 microseconds, respectively.

Although it is not visible in Figure 29, only 88,250 transactions (0.078% of the submitted workload) had their data recovered by the on-demand technique (orange dots). The remaining transactions (over 113 million, that is, 99,92%) accessed data that had been previously loaded into the memory (blue dots). To overcome this problem, Figure 30 is a boxplot that allows to better observe the dispersion and distribution of the transaction latency values. The boxplot only registers the latency values of transactions that were during the recovery.

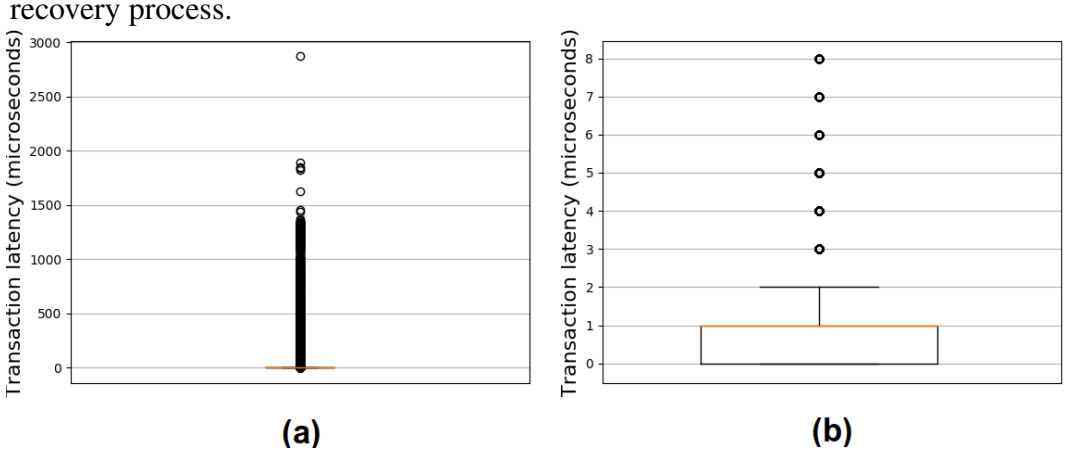
Figure 30 (a) shows that most latency values are close to zero and the outliers have

Figure 29 – Transaction latency scatterplot graph during instant recovery experiment.



very high values. However, the quartiles can not be observed. To solve this problem, Figure 30 (b) makes a close-up representation of Figure 30 (a) to allow the visualization of the quartiles. Looking carefully at Figure 30 (b), one may see that actually most latency values are close to 1 microsecond instead of zero, as previously thought in Figure 30 (a). That value is closer to the average transaction latency during recovery (0.9908 microseconds). In addition, values higher than 2 microseconds are considered outliers. Consequently, all restore latency values have been considered outliers since the lowest was 43 microseconds.

Figure 30 – Dispersion and distribution of transaction latency during the instant recovery process.



The results of the restore latency experiment indicate that some transactions had a high latency rate due to restore latency, a side-effect of the on-demand recovery implemented by *MM-DIRECT*. Nonetheless, we have noted that the amount of transactions with restore latency tends to be small (0.078% of the transactions). Furthermore, those transactions were responsible

for requesting on-demand recovery of 88% of the data accessed by the workload.

5.3 Checkpoint experiments

Next, we evaluate the proposed checkpoint technique. The experiments performed in this section measured checkpoint efficiency and overhead (provoked by checkpoint generation). Five experiments have been performed using different checkpoints: TuCC, TuCC-MFU, TUCC with interruption, TuCC-MFU with interruption, and no checkpoint. Interruption means a system failure during the checkpoint generation process.

For the experiments, a workload is submitted during normal system processing, and after 4,500 seconds, the system is shut down to simulate a failure. This timeframe was chosen to assure that at least one checkpoint had been produced before the failure. At the database restart, the workload is submitted again, as soon as the recovery process is triggered. Transaction throughput was measured during the experiments to evaluate the impact of the checkpoint technique on system performance. Additionally, recovery time for each checkpoint technique has been quantified.

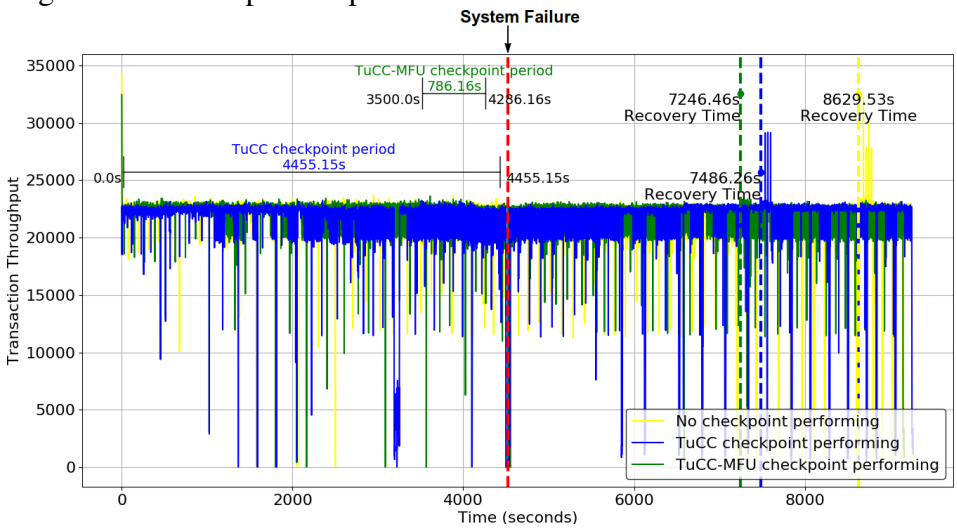
As mentioned in Section 5.1, the utilized database was generated by running the workload 20 times. Therefore, there already exists a 62.7GB sequential log file, which corresponds to $1.8x10^9$ log record. The indexed log file was generated along with the sequential log using the recovery mechanism proposed in this work. In this way, TuCC generation has been executed on that initial log file. For that reason, in the experiments, we considered the TuCC generation process to have begun at time zero (see Figure 31). The TuCC-MFU execution started later than TuCC, but before the system failure with enough time to run completely. This start time was chosen so that too many log records were not generated after the checkpoint ended, and consequently the work performed by just one checkpoint execution could be observed after the failure. Interrupted checkpoint experiments were started too close to system failure so that they could not run completely. Consequently, they were stopped by the system failure.

Figures 31, 32, and 33 bring the results of the checkpoint experiments. The vertical dashed lines indicate the final recovery time of the experiments, which are represented by the colors. The blue part represents the experiment with a TuCC execution that started at time zero. The green part represents the experiment with a TuCC-MFU execution that started after 3,500 seconds of transaction processing. The orange and purple parts represent TuCC and TuCC-MFU checkpoints, respectively, which started after 4,200 seconds of transaction processing and were

interrupted by a system failure. The yellow part represents the experiment without checkpoints.

Figure 31 compares TuCC checkpoint, TuCC-MFU checkpoint, and no-checkpoint. The transaction throughput rates are similar in the three experiments, i.e., the checkpoint generation process does not interfere in transaction throughput rates. As expected, the database recovery process is faster with the proposed checkpoint techniques. This happens because there are fewer log records to process during recovery. TuCC-MFU was the most efficient checkpoint technique because it supports a faster database recovery. It is important to remember that TuCC-MFU only checkpoints the most used tuples, whereas TuCC checkpoints all tuples in the database, including he tuples that were not updated.

Figure 31 – Checkpoint experiments.

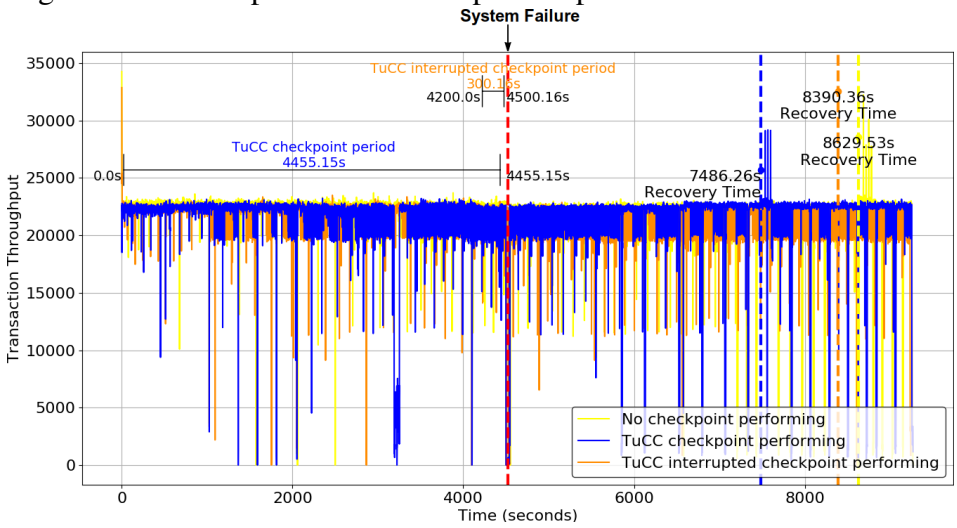


It is important to emphasize that even if TuCC and TuCC-MFU checkpoint processes are interrupted, they yield a consistent database state. Figure 32 compares TuCC without interruption (blue line), TuCC with interruption (orange line), and no-checkpoint (yellow line).

The blue and yellow line experiments in Figure 32 are the same blue and yellow line experiments shown in Figure 31. The orange line checkpoint experiment could not run completely because it was stopped by the system failure. The Figure 32 results show that although the TuCC process had been interrupted by the system failure, it assures a faster database recovery than when there is no checkpoint. However, as the TuCC process was interrupted, it could not checkpoint all necessary tuples. Consequently, it could not deliver a database recovery as fast as TuCC without interruption.

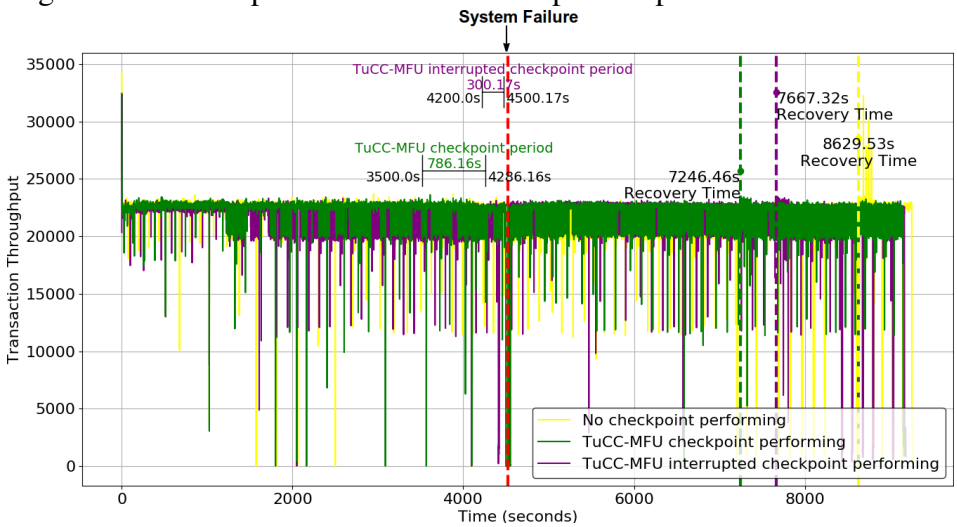
Figure 33 compares TuCC-MFU without interruption (green), TuCC-MFU with interruption (purple line), and no-checkpoint (yellow line). The green and yellow line experiments

Figure 32 – Interrupted TuCC checkpoint experiment.



in Figure 33 are the same green and yellow line experiments shown in Figure 31. Same as in the orange line (Figure 32), the purple line experiment checkpoint could not run completely because it was stopped by the system failure. The experiments' results shown in Figure 33 for TuCC-MFU with interruption presents a similar behavior as the results depicted in Figure 32 for TuCC with interruption. The interrupted TuCC-MFU processing experiment had faster recovery than the no-checkpoint experiment, but it could not checkpoint all necessary tuples and, consequently, it had a slower recovery than the TuCC-MFU without interruption.

Figure 33 – Interrupted TuCC-MFU checkpoint experiment.



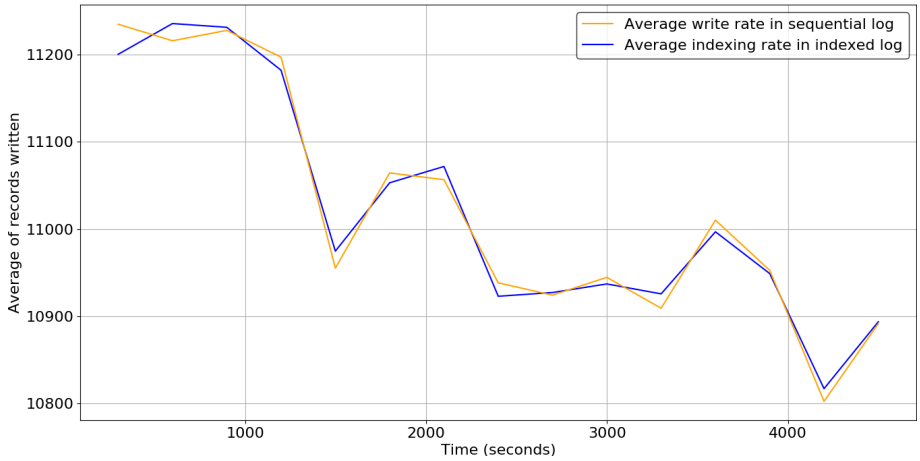
5.4 Log files' write bandwidth experiments

Asynchronous insertion of records in the indexed log is the key to *MM-DIRECT* achieving a high transaction throughput rate even during database recovery, as it is shown in Section 5.2. The sequential log write operation is potentially faster than the indexed log write operation, as discussed in Section 4.3. In other words, the number of write operations executed per second (write bandwidth) in a sequential log file is much higher than in the proposed indexed log file, with a B^+-tree structure. In this sense, the experiments presented in this section measure and compare write bandwidth in the sequential log file and the indexed log file.

In this section experiment, the workload was submitted during normal system processing of the *MM-DIRECT* (IRAIL scenario). No system failures were simulated in this experiment, as we want to observe the impact of log file write bandwidth on transaction processing only. In this way, first, the database was loaded entirely into memory, and then the workload was triggered to, from that moment on, start measuring the rate of writing to the log files.

In Figure 34, the orange and blue lines represent the write bandwidth in the sequential log and in the indexed log, respectively. Observe that both bandwidths are quite similar. Such a surprising result is a consequence of the buffer mechanism implemented by *MM-DIRECT* (see Section 4.3). Thus, the proposed buffer mechanism mitigates the problem of a potential low write bandwidth in the indexed log file. Furthermore, the two lines of Figure 34 tend downwards, i.e., the number of writes decreases with time. This is due to the behavior of the benchmark that performed more writes at the beginning of the experiments.

Figure 34 – Write bandwidth in sequential and in indexed log files.



It is important to note that the indexed log behavior shown in this section experiments avoids a long log tail and, consequently, a downtime after a system failure. A short log tail means

that few records must be inserted into the indexed log before the recovery process begins, as discussed in Section 4.4. Thus, very little time must be spent before the instant recovery begins, i.e., downtime is insignificant, as shown in Section 5.2 experiment results.

5.5 CPU and memory overhead experiments

Since the disk I/O is not the main source of overhead for MMDBs, CPU and memory usage can become a bottleneck for these systems. Thus, this section investigates the CPU and memory overheads imposed by *MM-DIRECT* recovery mechanism compared to the Standard Recovery mechanism. The experiments used *Top* (NORDBY *et al.*, ; PALAKOLLU, 2021), a program that gives continual reports about the state of the Linux system, including a list of the processes using the CPU and memory.

This section's experiments are similar to the previous section's experiment (Section 5.4), in which no system failure was triggered, only transaction processing was observed. However, the CPU and memory usage of the database system is measured rather than the log file write bandwidth. The experiments were performed both in *MM-DIRECT* (IRAIL scenario) and in Standard Recovery (SRSL scenario). In each experiment, first, the database was loaded entirely into memory, and then the workload was triggered to start measuring the CPU and memory usage.

Figure 35 represents the average CPU usage (in percentage) over small time intervals. One may observe that *MM-DIRECT* (blue line) uses more CPU than default recovery (yellow line). This is comprehensible as *MM-DIRECT* handles two log files while standard recovery only handles one log file. These results refer to the CPU usage in relation to the power of 1 core. For example, if the CPU usage was over 100%, it would mean that more than one core was busy. That way, the system would be overloaded, i.e., there would be more work to be put into a core than is physically manageable. These results show that the *MM-DIRECT* recovery mechanism does not impose an overhead on the system.

Figure 36 represents the average memory usage (in megabytes) over small time intervals. The results show that *MM-DIRECT* (blue line) needs a little more memory than default recover (yellow line). This is explained by the fact that *MM-DIRECT* handles two log files. However, *MM-DIRECT* performing does not impose any memory overhead on the system.

Figure 35 – Average CPU usage over small time intervals.

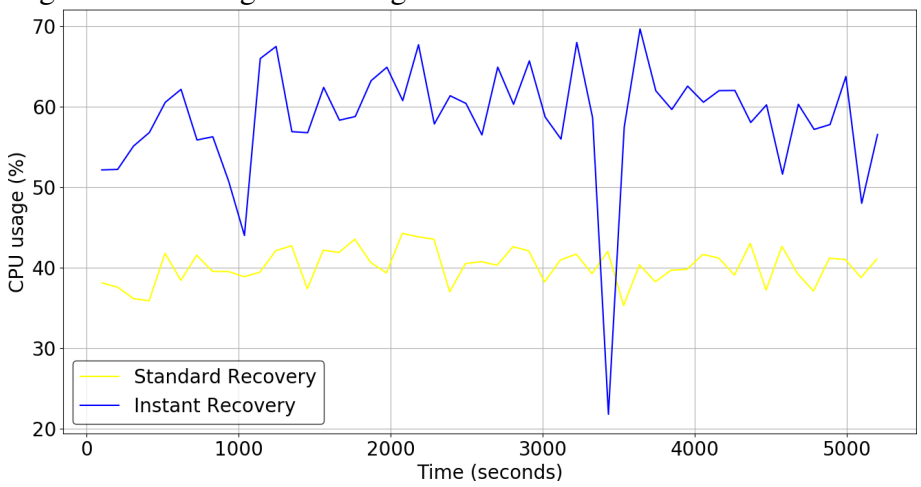
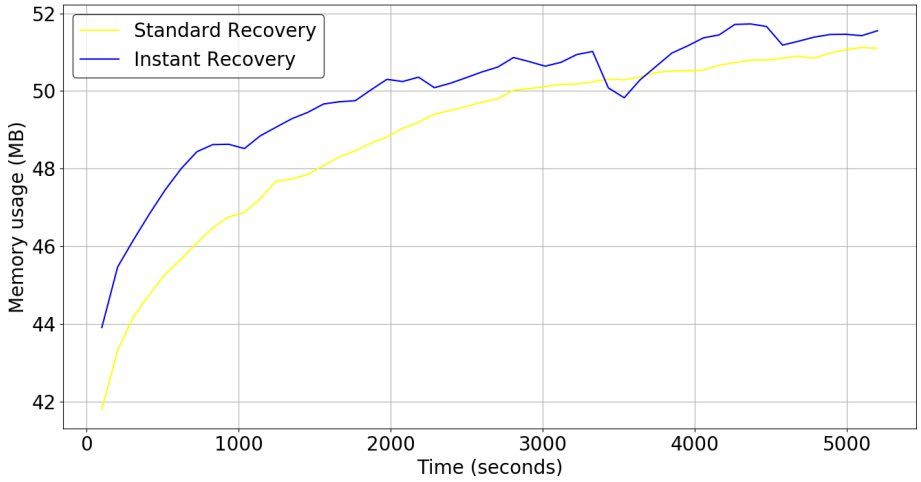


Figure 36 – Average memory usage over small time intervals.



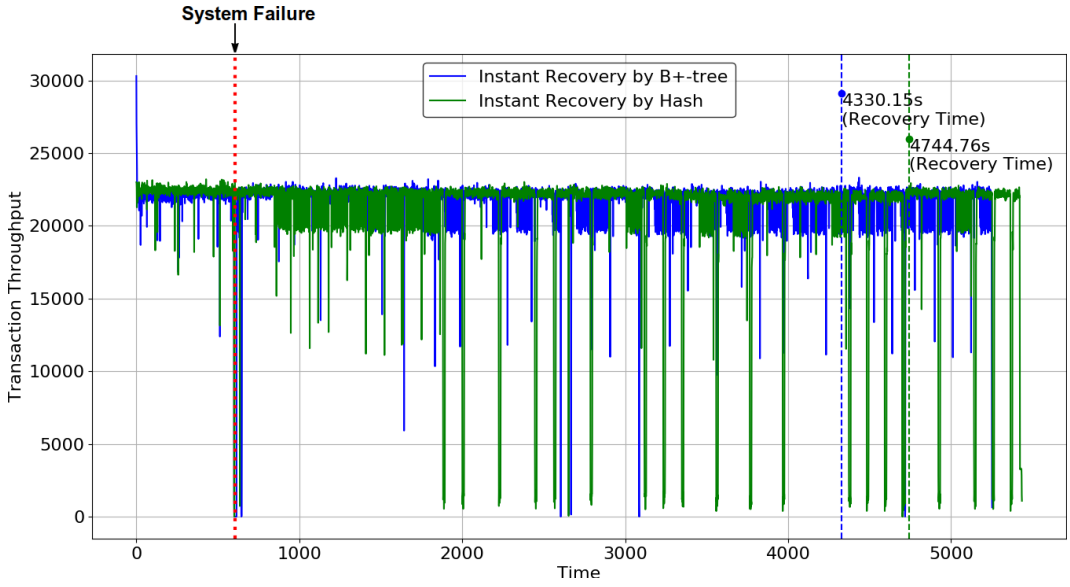
5.6 Recovery experiments using different index data structures

The *MM-DIRECT* indexed log structure was designed as a B$^+$-tree, but it is enable to support other indexes structures, such as Hash table. B$^+$-tree and Hash indexes have their trade-offs, as discussed in Section 4.2. Thus, this section compares and analyzes the behavior of the *MM-DIRECT* recovery mechanism using those index structures. The experiments were performed using two indexed log files: one with B$^+$-tree, and another with Hash. We performed an experiment for each indexed log file using *MM-DIRECT* (IRAIL scenario), similarly to the experiments in Figure 26. A workload was submitted for each experiment during normal system processing and the system was shut down after 600 seconds to simulate a failure. At the database restart, as soon as the recovery process was been triggered, the workload was submitted again. Transaction throughput was measured during the experiments. In addition, the recovery time, workload execution time, and the number of tuples recovered incrementally and on-demand were measured.

The B^+-tree indexed log file used in this experiment was the same used in the previous section experiments. Although the Hash indexed log can be constructed during normal database processing, similar to how the B^+-tree indexed log was generated, the Hash was constructed from the B^+-tree. In this way, the Hash and the B^+-tree should map to the same log records, i.e., they are equivalent. The Hash indexed log file was generated by that B^+-tree file, in which each Hash bucket with search key k contains the same log records, in the same order, as the B^+-tree leaf node whose search key is k. In this way, a search using a key k will return the same result on both the Hash and the B^+-tree.

Figure 37 shows the results of the experiment in this section. The green line experiment performed the *MM-DIRECT* recovery mechanism using Hash indexed log. The blue line is the same blue line experiment shown in Figure 26 that represents the *MM-DIRECT* recovery mechanism using B^+-tree indexed log. Looking at Figure 37, one may observe that B^+-tree and Hash provided a similar transaction throughput. However, B^+-tree recovered the database and ran the workload faster than Hash. These results occurred because incremental recovery accounted for most of the effort expended to recover the database in these experiments. Incremental recovery is done by traversing the index data structure. Traversing a B^+-tree is potentially faster than a Hash. The Hash recovered 84% of the database tuples incrementally. The B^+-tree recovered 85% of the database tuples incrementally.

Figure 37 – Transaction throughput during the instant recovery process through a B^+-tree and a Hash table.



5.7 Scalability experiments

We ran further experiments in which the proposed recovery strategy deals with different numbers of worker threads. The goal is to observe the behavior and performance of *MM-DIRECT* when the number of threads used increases. Four tests were performed with different numbers of worker threads: 4, 5, 6, and 8. The experiments used the same workload configuration as previous experiments, except they may use a different number of threads. However, as the number of worker threads increases, the number of clients increases, and consequently the number of requests increases. In this way, the workload increases as the number of threads grow. In these experiments, we measured the transaction throughput, recovery time, and CPU and memory usage in order to analyze the scalability of *MM-DIRECT*.

Figure 38 presents the average transaction throughput obtained during the full test in each experiment. The results show that the transaction throughput grows with the number of threads used, i.e., the system does not overload even if the workload increases. This experiments show that *MM-DIRECT* is able to effectively provide high transaction throughput rates.

Figure 38 – Average transaction throughput experiments using different worker threads.

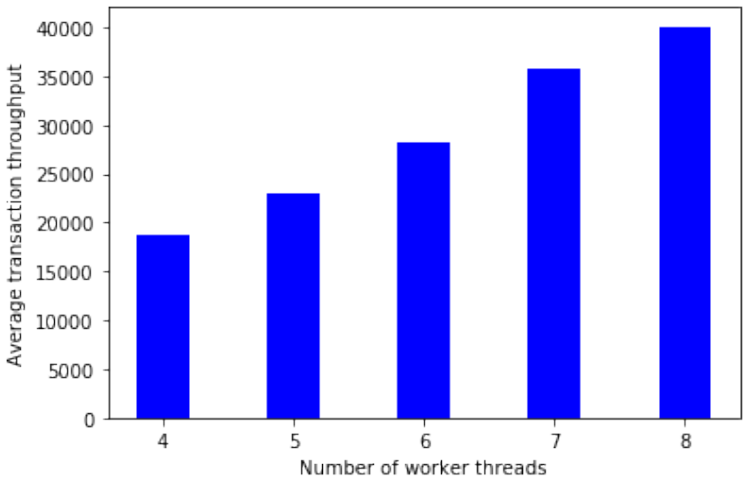


Figure 39 presents the average latency obtained during the recovery process in each experiment. The latency was measured only during the recovery process because of the restore latency. The restore latency is the main source of latency and occurs only during the recovery process, as discussed in Section 5.2. On the other hand, the latency is close to zero during normal transaction processing. Figure 39 shows that the average latency values are close in all experiments.

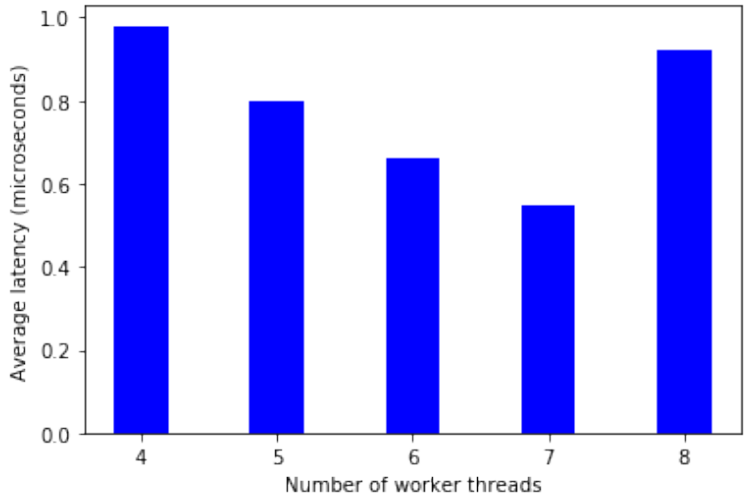


Figure 39 – Average latency experiments using different worker threads.

Figures 40 and 41 results show that the recovery time and full workload execution time, respectively, in the experiments are close. These results were already expected since the system transaction throughput increased in each experiment, as shown in Figure 38 experiments. Besides, the increase in worker threads did not influence the latency of the system, as shown in Figure 39 experiments.

Figures 42 and 43 show average CPU and memory usages of the database system, respectively, obtained during the full test in each experiment. One may observe that the proposed recovery mechanism does not overload the CPU and memory of the system in any of the experiments.

Figure 40 – Recovery time experiments using different worker threads.

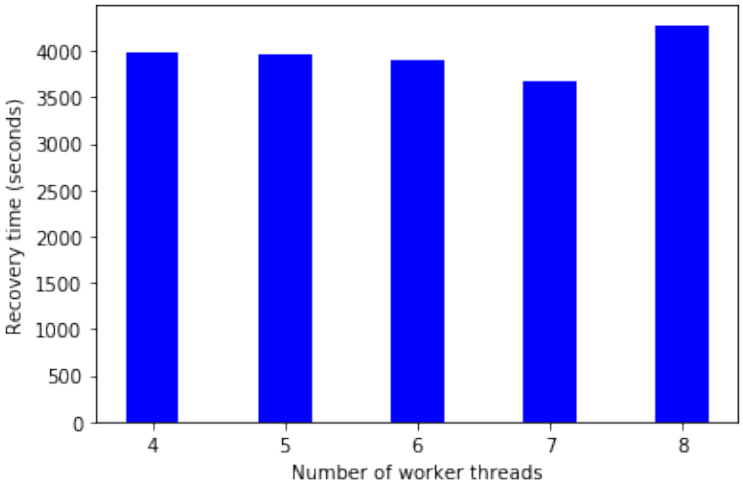


Figure 41 – Workload runtime experiments using different worker threads.

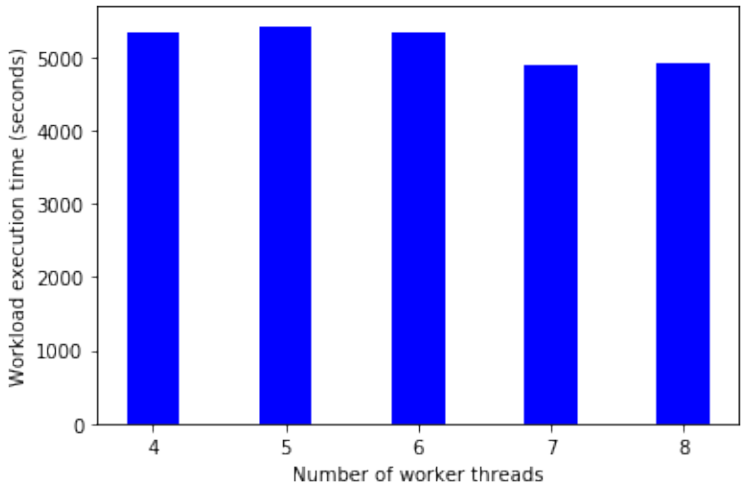


Figure 42 – Average CPU experiments usage using different worker threads.

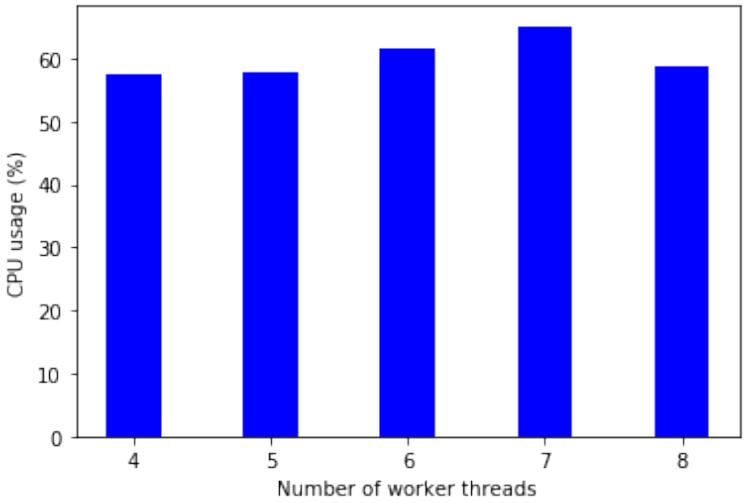
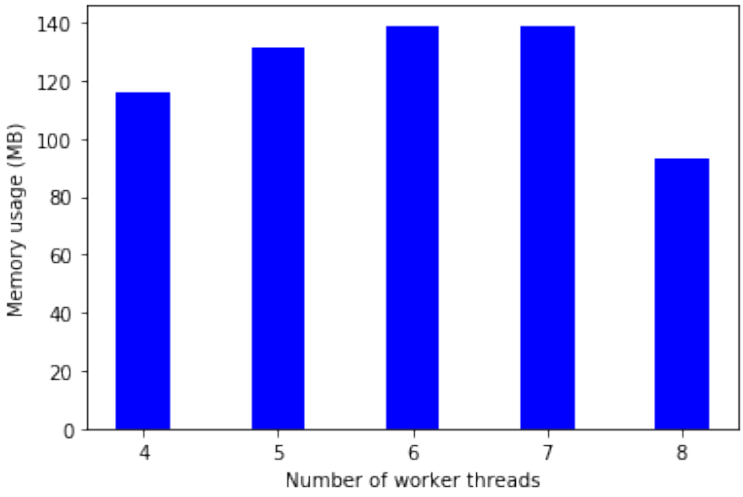


Figure 43 – Average memory experiments usage using different worker threads.



6 FUTURE WORKS

6.1 Usage of different workloads

Running different workloads on database systems can result in different system performance behaviors (PÖSS *et al.*, 2007; LO *et al.*, 1998; OH; LEE, 2004). Writes to the indexed log during database running and reads from the indexed log during database recovery can also influence system performance.

We intend to submit different workload patterns on the system during normal database running and during database recovery to evaluate the system performance.

6.2 Usage of other data structures to implement the log file

The indexed log for database instant recovery mechanism proposed in this work was designed to use a B^+-tree. However, the mechanism supports different index data structures, such as: Hash table.

We intend to evaluate the performance on the system using different indexed log data structures on different workload patterns. In this way, perhaps we may be able to identify whether a particular indexed log data structure is more appropriate for a particular workload pattern.

6.3 Log file partitioning

The log file corruption event can be caused due to multiple reasons, include: viruses or other malicious software, terminating system abnormally, input-output configuration problem, and hard drive issues (SCHUSTER, 2007; WINTER, 2018). The absence or corruption of the indexed log file makes the instant recovery proposed in this work impossible.

The indexed log proposed in this work is implemented through a B-tree in a single file. We plan to partition the indexed log file in multiple files in order to reduce the impact of a log file corruption. By partitioning the log into multiple files, not all files should be corrupted if a log file corruption event happens.

6.4 Parallel database recovery

The instant recovery mechanism proposed in this thesis uses only one thread to recover the data. In order to speed up recovery, a parallel recovery should be developed. For instance, indexed log partitioning could facilitate the development of a parallel database recovery mechanism. In this mechanism, different threads should load the data into memory in parallel from different log partitions. Each partition should be assigned to a thread (DIACONU *et al.*, 2013; ZHENG *et al.*, 2014; ROSENBLUM; OUSTERHOUT, 1992).

6.5 Intelligent checkpoint

In the current *MM-DIRECT* implementation, the DBA is responsible for choosing which checkpoint technique should be applied. For that the DBA should have in mind that TuCC-MFU has faster process execution and, on the other hand, TuCC allows faster reconstruction of a corrupted indexed log file. A future improvement in *MM-DIRECT* should implement a machine learning technique to chose the best checkpoint technique, for a given workload and database cardinality.

7 CONCLUSION

This thesis proposed an instant recovery mechanism for MMDBs. The mechanism allows new transactions to run concurrently to the recovery process. The mechanism 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.

BIBLIOGRAPHY

ABADI, D. J.; MADDEN, S.; HACHEM, N. Column-stores vs. row-stores: how different are they really? In: WANG, J. T. (Ed.). **Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008**. ACM, 2008. p. 967–980. Available in: https://doi.org/10.1145/1376616.1376712. Access in: Tue, 06 Nov 2018 11:07:38 +0100.

AGRAWAL, R.; CAREY, M. J.; LIVNY, M. Concurrency control performance modeling: Alternatives and implications. **ACM Transactions on Database Systems (TODS)**, v. 12, n. 4, p. 609–654, 1987. Available in: https://doi.org/10.1145/32204.32220. Access in: Tue, 06 Nov 2018 12:51:48 +0100.

AGRAWAL, R.; DEWITT, D. J. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. **ACM Transactions on Database Systems (TODS)**, v. 10, n. 4, p. 529–564, 1985. Available in: https://doi.org/10.1145/4879.4958. Access in: Tue, 06 Nov 2018 12:51:48 +0100.

AGRAWAL, R.; JAGADISH, H. V. Recovery algorithms for database machines with nonvolatile main memory. In: BORAL, H.; FAUDEMAY, P. (Ed.). **Database Machines, Sixth International Workshop, IWDM '89, Deauville, France, June 19-21, 1989, Proceedings**. Springer, 1989. (Lecture Notes in Computer Science, v. 368), p. 269–285. Available in: https://doi.org/10.1007/3-540-51324-8_41. Access in: Tue, 14 May 2019 10:00:42 +0200.

AILAMAKI, A.; DEWITT, D. J.; HILL, M. D.; WOOD, D. A. Dbmss on a modern processor: Where does time go? In: ATKINSON, M. P.; ORLOWSKA, M. E.; VALDURIEZ, P.; ZDONIK, S. B.; BRODIE, M. L. (Ed.). **VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK**. Morgan Kaufmann, 1999. p. 266–277. Available in: http://www.vldb.org/conf/1999/P28.pdf. Access in: Thu, 12 Mar 2020 11:33:53 +0100.

ALBUTIU, M.; KEMPER, A.; NEUMANN, T. Massively parallel sort-merge joins in main memory multi-core database systems. **Proceedings of the VLDB Endowment**, v. 5, n. 10, p. 1064–1075, 2012. Available in: http://vldb.org/pvldb/vol5/p1064_martina-cezaraalbutiu_vldb2012.pdf. Access in: Sat, 25 Apr 2020 13:58:35 +0200.

ALEXIOU, K.; KOSSMANN, D.; LARSON, P. Adaptive range filters for cold data: Avoiding trips to siberia. **Proceedings of the VLDB Endowment**, v. 6, n. 14, p. 1714–1725, 2013. Available in: http://www.vldb.org/pvldb/vol6/p1714-kossmann.pdf. Access in: Sat, 25 Apr 2020 13:58:49 +0200.

ARULRAJ, J.; PAVLO, A. How to build a non-volatile memory database management system. In: SALIHOGLU, S.; ZHOU, W.; CHIRKOVA, R.; YANG, J.; SUCIU, D. (Ed.). **Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017**. ACM, 2017. p. 1753–1758. Available in: https://doi.org/10.1145/3035918.3054780. Access in: Tue, 06 Nov 2018 11:07:39 +0100.

ARULRAJ, J.; PAVLO, A.; DULLOOR, S. Let's talk about storage & recovery methods for non-volatile memory database systems. In: SELLIS, T. K.; DAVIDSON, S. B.; IVES, Z. G. (Ed.). **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015**. ACM, 2015. p. 707–722.

Available in: https://doi.org/10.1145/2723372.2749441. Access in: Tue, 06 Nov 2018 11:07:38 +0100.

ARULRAJ, J.; PAVLO, A.; MENON, P. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In: ÖZCAN, F.; KOUTRIKA, G.; MADDEN, S. (Ed.). **Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016**. ACM, 2016. p. 583–598. Available in: https://doi.org/10.1145/2882903.2915231. Access in: Wed, 14 Nov 2018 10:56:20 +0100.

ARULRAJ, J.; PERRON, M.; PAVLO, A. Write-behind logging. **Proceedings of the VLDB Endowment**, v. 10, n. 4, p. 337–348, 2016. Available in: http://www.vldb.org/pvldb/vol10/p337-arulraj.pdf. Access in: Sat, 25 Apr 2020 13:58:33 +0200.

BAILIS, P.; FEKETE, A. D.; FRANKLIN, M. J.; GHODSI, A.; HELLERSTEIN, J. M.; STOICA, I. Coordination avoidance in database systems. **Proceedings of the VLDB Endowment**, v. 8, n. 3, p. 185–196, 2014. Available in: http://www.vldb.org/pvldb/vol8/p185-bailis.pdf. Access in: Thu, 27 Aug 2020 11:23:09 +0200.

BAULIER, J.; BOHANNON, P.; GOGATE, S.; GUPTA, C.; HALDAR, S.; JOSHI, S.; KHIVESERA, A.; KORTH, H. F.; MCILROY, P.; MILLER, J.; NARAYAN, P. P. S.; NEMETH, M.; RASTOGI, R.; SESHADRI, S.; SILBERSCHATZ, A.; SUDARSHAN, S.; WILDER, M.; WEI, C. Datablitz storage manager: Main memory database performance for critical applications. In: DELIS, A.; FALOUTSOS, C.; GHANDEHARIZADEH, S. (Ed.). **SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA**. ACM Press, 1999. p. 519–520. Available in: https://doi.org/10.1145/304182.304239. Access in: Tue, 06 Nov 2018 11:07:37 +0100.

BAYER, R.; SCHKOLNICK, M. Concurrency of operations on b-trees. **Acta Informatica**, v. 9, p. 1–21, 1977. Available in: https://doi.org/10.1007/BF00263762. Access in: Sun, 21 Jun 2020 17:37:58 +0200.

BEHRAVESH, R.; CORONADO, E.; RIGGIO, R. Performance evaluation on virtualization technologies for NFV deployment in 5g networks. In: JACQUENET, C.; TURCK, F. D.; CHEMOUIL, P.; ESPOSITO, F.; FESTOR, O.; CERRONI, W.; SECCI, S. (Ed.). **5th IEEE Conference on Network Softwarization, NetSoft 2019, Paris, France, June 24-28, 2019**. IEEE, 2019. p. 24–29. Available in: https://doi.org/10.1109/NETSOFT.2019.8806664. Access in: Tue, 29 Dec 2020 18:41:15 +0100.

BERKELEY DB DOCUMENTATION. **Berkeley DB Programmer's Reference Guide**. 2020. Available in: https://docs.oracle.com/cd/E17276_01/html/programmer_reference. Access in: October 06, 2020.

BERNSTEIN, P. A.; GOODMAN, N. Concurrency control in distributed database systems. **ACM Computing Surveys (CSUR)**, v. 13, n. 2, p. 185–221, 1981. Available in: https://doi.org/10.1145/356842.356846. Access in: Tue, 06 Nov 2018 12:50:47 +0100.

BERNSTEIN, P. A.; HADZILACOS, V.; GOODMAN, N. **Concurrency Control and Recovery in Database Systems**. Addison-Wesley, 1987. ISBN 0-201-10715-5. Available in: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx. Access in: Wed, 10 Jul 2019 10:47:07 +0200.

BEZ, R.; CAMERLENGHI, E.; MODELLI, A.; VISCONTI, A. Introduction to flash memory. **Proc. IEEE**, v. 91, n. 4, p. 489–502, 2003. Available in: https://doi.org/10.1109/JPROC.2003.811702. Access in: Thu, 23 Sep 2021 11:46:38 +0200.

BINNA, R.; ZANGERLE, E.; PICHL, M.; SPECHT, G.; LEIS, V. HOT: A height optimized trie index for main-memory database systems. In: DAS, G.; JERMAINE, C. M.; BERNSTEIN, P. A. (Ed.). **Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018**. ACM, 2018. p. 521–534. Available in: https://doi.org/10.1145/3183713.3196896. Access in: Wed, 21 Nov 2018 12:44:08 +0100.

BITTON, D.; GRAY, J. Disk shadowing. In: BANCILHON, F.; DEWITT, D. J. (Ed.). **Fourteenth International Conference on Very Large Data Bases, August 29 - September 1, 1988, Los Angeles, California, USA, Proceedings**. Morgan Kaufmann, 1988. p. 331–338. Available in: http://www.vldb.org/conf/1988/P331.PDF. Access in: Mon, 06 Nov 2017 16:35:11 +0100.

BITTON, D.; HANRAHAN, M.; TURBYFILL, C. Performance of complex queries in main memory database systems. In: **Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA**. IEEE Computer Society, 1987. p. 72–81. Available in: https://doi.org/10.1109/ICDE.1987.7272358. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

BORAL, H.; ALEXANDER, W.; CLAY, L.; COPELAND, G. P.; DANFORTH, S.; FRANKLIN, M. J.; HART, B. E.; SMITH, M. G.; VALDURIEZ, P. Prototyping bubba, A highly parallel database system. **IEEE Transactions on Knowledge and Data**, v. 2, n. 1, p. 4–24, 1990. Available in: https://doi.org/10.1109/69.50903. Access in: Sat, 20 May 2017 00:24:22 +0200.

BRONEVETSKY, G.; FERNANDES, R.; MARQUES, D.; PINGALI, K.; STODGHILL, P. Recent advances in checkpoint/recovery systems. In: **20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece**. IEEE, 2006. Available in: https://doi.org/10.1109/IPDPS.2006.1639575. Access in: Wed, 16 Oct 2019 14:14:51 +0200.

CAO, W.; SAHIN, S.; LIU, L.; BAO, X. Evaluation and analysis of in-memory key-value systems. In: PU, C.; FOX, G. C.; DAMIANI, E. (Ed.). **2016 IEEE International Congress on Big Data, San Francisco, CA, USA, June 27 - July 2, 2016**. IEEE Computer Society, 2016. p. 26–33. Available in: https://doi.org/10.1109/BigDataCongress.2016.13. Access in: Wed, 16 Oct 2019 14:14:55 +0200.

CAPPELLETTI, P.; GOLLA, C.; OLIVO, P.; ZANONI, E. **Flash memories**. [*S. l.*]: Springer Science & Business Media, 2013.

CAREY, M. J.; STONEBRAKER, M. The performance of concurrency control algorithms for database management systems. In: DAYAL, U.; SCHLAGETER, G.; SENG, L. H. (Ed.). **Tenth International Conference on Very Large Data Bases, August 27-31, 1984, Singapore, Proceedings**. Morgan Kaufmann, 1984. p. 107–118. Available in: http://www.vldb.org/conf/1984/P107.PDF. Access in: Tue, 07 Nov 2017 06:51:45 +0100.

CHA, S. K.; SONG, C. P*time: Highly scalable OLTP DBMS for managing update-intensive stream workload. In: NASCIMENTO, M. A.; ÖZSU, M. T.; KOSSMANN, D.; MILLER, R. J.; BLAKELEY, J. A.; SCHIEFER, K. B. (Ed.). **(e)Proceedings of the Thirtieth**

International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004. Morgan Kaufmann, 2004. p. 1033–1044. Available in: http://www.vldb.org/conf/2004/IND2P2.PDF. Access in: Fri, 07 Jun 2019 12:44:01 +0200.

CHANG, F.; DEAN, J.; GHEMAWAT, S.; HSIEH, W. C.; WALLACH, D. A.; BURROWS, M.; CHANDRA, T.; FIKES, A.; GRUBER, R. E. Bigtable: A distributed storage system for structured data. **ACM Trans. Comput. Syst.**, v. 26, n. 2, p. 4:1–4:26, 2008. Available in: https://doi.org/10.1145/1365815.1365816. Access in: Sun, 02 Jun 2019 21:08:58 +0200.

CHATZISTERGIOU, A.; CINTRA, M.; VIGLAS, S. D. REWIND: recovery write-ahead system for in-memory non-volatile data-structures. **Proceedings of the VLDB Endowment**, v. 8, n. 5, p. 497–508, 2015. Available in: http://www.vldb.org/pvldb/vol8/p497-chatzistergiou.pdf. Access in: Sat, 25 Apr 2020 13:59:11 +0200.

CHEN, J.; JINDEL, S.; WALZER, R.; SEN, R.; JIMSHELEISHVILLI, N.; ANDREWS, M. The memsql query optimizer: A modern optimizer for real-time analytics in a distributed database. **Proceedings of the VLDB Endowment**, v. 9, n. 13, p. 1401–1412, 2016. Available in: http://www.vldb.org/pvldb/vol9/p1401-chen.pdf. Access in: Sat, 25 Apr 2020 13:58:37 +0200.

CHEN, S.; GIBBONS, P. B.; MOWRY, T. C. Improving index performance through prefetching. In: MEHROTRA, S.; SELLIS, T. K. (Ed.). **Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001**. ACM, 2001. p. 235–246. Available in: https://doi.org/10.1145/375663.375688. Access in: Tue, 06 Nov 2018 11:07:39 +0100.

CHEN, S.; GIBBONS, P. B.; NATH, S. Rethinking database algorithms for phase change memory. In: **CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings**. www.cidrdb.org, 2011. p. 21–31. Available in: http://cidrdb.org/cidr2011/Papers/CIDR11_Paper3.pdf. Access in: Thu, 12 Mar 2020 11:32:37 +0100.

CHEN, S.; GIBBONS, P. B.; NATH, S. Rethinking database algorithms for phase change memory. In: **CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings**. www.cidrdb.org, 2011. p. 21–31. Available in: http://cidrdb.org/cidr2011/Papers/CIDR11_Paper3.pdf. Access in: Thu, 12 Mar 2020 11:32:37 +0100.

CHENG, J. M.; LOOSELEY, C. R.; SHIBAMIYA, A.; WORTHINGTON, P. S. IBM database 2 performance: Design, implementation, and tuning. **IBM Systems Journal**, v. 23, n. 2, p. 189–210, 1984. Available in: https://doi.org/10.1147/sj.232.0189. Access in: Fri, 13 Mar 2020 14:38:36 +0100.

CHHUGANI, J.; NGUYEN, A. D.; LEE, V. W.; MACY, W.; HAGOG, M.; CHEN, Y.; BARANSI, A.; KUMAR, S.; DUBEY, P. Efficient implementation of sorting on multi-core SIMD CPU architecture. **Proceedings of the VLDB Endowment**, v. 1, n. 2, p. 1313–1324, 2008. Available in: http://www.vldb.org/pvldb/vol1/1454171.pdf. Access in: Sat, 25 Apr 2020 13:59:34 +0200.

CHOI, J.; LIAN, R.; LI, Z.; CANIS, A.; ANDERSON, J. H. Accelerating memcached on AWS cloud fpgas. In: **Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, HEART 2018, Toronto, ON, Canada, June**

20-22, 2018. ACM, 2018. p. 2:1–2:8. Available in: https://doi.org/10.1145/3241793.3241795. Access in: Fri, 17 Jan 2020 17:11:14 +0100.

CLARK, B. E.; CORRIGAN, M. J. Application system/400 performance characteristics. **IBM Systems Journal**, v. 28, n. 3, p. 407–423, 1989. Available in: https://doi.org/10.1147/sj.283.0407. Access in: Fri, 13 Mar 2020 14:38:42 +0100.

COBURN, J.; BUNKER, T.; SCHWARZ, M.; GUPTA, R.; SWANSON, S. From ARIES to MARS: transaction support for next-generation, solid-state drives. In: KAMINSKY, M.; DAHLIN, M. (Ed.). **ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013**. ACM, 2013. p. 197–212. Available in: https://doi.org/10.1145/2517349.2522724. Access in: Tue, 06 Nov 2018 16:59:32 +0100.

COMER, D. The ubiquitous b-tree. **ACM Comput. Surv.**, v. 11, n. 2, p. 121–137, 1979. Available in: https://doi.org/10.1145/356770.356776. Access in: Wed, 14 Nov 2018 10:31:47 +0100.

COPELAND, G. P.; KHOSHAFIAN, S. A decomposition storage model. In: NAVATHE, S. B. (Ed.). **Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 28-31, 1985**. ACM Press, 1985. p. 268–279. Available in: https://doi.org/10.1145/318898.318923. Access in: Sun, 02 Jun 2019 21:17:44 +0200.

CRUS, R. A. Data recovery in IBM database 2. **IBM Systems Journal**, v. 23, n. 2, p. 178–188, 1984. Available in: https://doi.org/10.1147/sj.232.0178. Access in: Fri, 13 Mar 2020 14:38:17 +0100.

CUI, B.; OOI, B. C.; SU, J.; TAN, K.-L. Contorting high dimensional data for efficient main memory knn processing. In: **Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2003. (SIGMOD '03), p. 479–490. ISBN 158113634X. Available in: https://doi.org/10.1145/872757.872815. Access in: July 22, 2020.

CUI, B.; OOI, B. C.; SU, J.; TAN, K. Main memory indexing: The case for bd-tree. **IEEE Transactions on Knowledge and Data**, v. 16, n. 7, p. 870–874, 2004. Available in: https://doi.org/10.1109/TKDE.2004.1318568. Access in: Tue, 19 Sep 2017 08:02:49 +0200.

DEBRABANT, J.; ARULRAJ, J.; PAVLO, A.; STONEBRAKER, M.; ZDONIK, S. B.; DULLOOR, S. A prolegomenon on OLTP database systems for non-volatile memory. In: BORDAWEKAR, R.; LAHIRI, T.; GEDIK, B.; LANG, C. A. (Ed.). **International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2014, Hangzhou, China, September 1, 2014**. [*S. n.*], 2014. p. 57–63. Available in: http://www.adms-conf.org/2014/adms14_debrabant.pdf. Access in: Thu, 12 Mar 2020 11:33:53 +0100.

DEBRABANT, J.; PAVLO, A.; TU, S.; STONEBRAKER, M.; ZDONIK, S. B. Anti-caching: A new approach to database management system architecture. **Proceedings of the VLDB Endowment**, v. 6, n. 14, p. 1942–1953, 2013. Available in: http://www.vldb.org/pvldb/vol6/p1942-debrabant.pdf. Access in: Sat, 25 Apr 2020 13:59:03 +0200.

DEWITT, D. J.; GHANDEHARIZADEH, S.; SCHNEIDER, D. A.; BRICKER, A.; HSIAO, H.; RASMUSSEN, R. The gamma database machine project. **IEEE Transactions on Knowledge and Data**, v. 2, n. 1, p. 44–62, 1990. Available in: https://doi.org/10.1109/69.50905. Access in: Wed, 14 Nov 2018 10:17:27 +0100.

DEWITT, D. J.; KATZ, R. H.; OLKEN, F.; SHAPIRO, L. D.; STONEBRAKER, M.; WOOD, D. A. Implementation techniques for main memory database systems. In: YORMARK, B. (Ed.). **SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984**. ACM Press, 1984. p. 1–8. Available in: https://doi.org/10.1145/602259.602261. Access in: Mon, 25 Mar 2019 12:12:16 +0100.

DIACONU, C.; FREEDMAN, C.; ISMERT, E.; LARSON, P.; MITTAL, P.; STONECIPHER, R.; VERMA, N.; ZWILLING, M. Hekaton: SQL server's memory-optimized OLTP engine. In: ROSS, K. A.; SRIVASTAVA, D.; PAPADIAS, D. (Ed.). **Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013**. ACM, 2013. p. 1243–1254. Available in: https://doi.org/10.1145/2463676.2463710. Access in: Tue, 06 Nov 2018 11:07:38 +0100.

DRAGOJEVIC, A.; NARAYANAN, D.; CASTRO, M.; HODSON, O. Farm: Fast remote memory. In: MAHAJAN, R.; STOICA, I. (Ed.). **Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014**. USENIX Association, 2014. p. 401–414. Available in: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevic. Access in: Tue, 02 Feb 2021 08:05:46 +0100.

DRISKILL-SMITH, A. Latest advances and future prospects of stt-ram. In: **Non-Volatile Memories Workshop**. [*S. n.*], 2010. p. 11–13. Available in: http://nvmw.eng.ucsd.edu/2010/documents/Driskill-Smith_Alexander.pdf. Access in: July 22, 2020.

EICH, M. H. Main memory database recovery. In: **Proceedings of the Fall Joint Computer Conference, November 2-6, 1986, Dallas, Texas, USA**. IEEE Computer Society, 1986. p. 1226–1232. Available in: https://dl.acm.org/doi/pdf/10.5555/324493.325092. Access in: Fri, 29 Sep 2017 14:35:52 +0200.

EICH, M. H. A classification and comparison of main memory database recovery techniques. In: **Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA**. IEEE Computer Society, 1987. p. 332–339. Available in: https://doi.org/10.1109/ICDE.1987.7272398. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

EICH, M. H. MARS: the design of a main memory database machine. In: KITSUREGAWA, M.; TANAKA, H. (Ed.). **Database Machines and Knowledge Base Machines, 5th International Workshop on Database Machines, Tokyo, Japan, 1987, Proceedings**. Kluwer, 1987. (The Kluwer International Series in Engineering and Computer Science, v. 43), p. 325–338. Available in: https://doi.org/10.1007/978-1-4613-1679-4_24. Access in: Fri, 19 May 2017 01:25:51 +0200.

ELDAWY, A.; LEVANDOSKI, J. J.; LARSON, P. Trekking through siberia: Managing cold data in a memory-optimized database. **Proceedings of the VLDB Endowment**, v. 7, n. 11, p. 931–942, 2014. Available in: http://www.vldb.org/pvldb/vol7/p931-eldawy.pdf. Access in: Mon, 15 Jun 2020 16:56:23 +0200.

FAERBER, F.; KEMPER, A.; LARSON, P.; LEVANDOSKI, J. J.; NEUMANN, T.; PAVLO, A. Main memory database systems. **Foundations and Trends in Databases**, v. 8, n. 1-2, p. 1–130, 2017. Available in: https://doi.org/10.1561/1900000058. Access in: Sat, 25 Apr 2020 14:00:55 +0200.

FAN, B.; ANDERSEN, D. G.; KAMINSKY, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In: FEAMSTER, N.; MOGUL, J. C. (Ed.). **Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013**. USENIX Association, 2013. p. 371–384. Available in: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan. Access in: Tue, 02 Feb 2021 08:05:05 +0100.

FÄRBER, F.; CHA, S. K.; PRIMSCH, J.; BORNHÖVD, C.; SIGG, S.; LEHNER, W. SAP HANA database: data management for modern business applications. **ACM Sigmod Record**, v. 40, n. 4, p. 45–51, 2011. Available in: https://doi.org/10.1145/2094114.2094126. Access in: Fri, 06 Mar 2020 21:56:32 +0100.

FÄRBER, F.; MAY, N.; LEHNER, W.; GROSSE, P.; MÜLLER, I.; RAUHE, H.; DEES, J. The SAP HANA database – an architecture overview. **IEEE Database Engineering Bulletin**, v. 35, n. 1, p. 28–33, 2012. Available in: http://sites.computer.org/debull/A12mar/hana.pdf. Access in: Tue, 10 Mar 2020 16:23:50 +0100.

FORK. **Fork (system call)**. 2020. Accessed in December 1, 2020. Available in: https://en.wikipedia.org/wiki/Fork_(system_call). Access in: Decemberay 1, 2020.

FRANASZEK, P. A.; ROBINSON, J. T. Limitations of concurrency in transaction processing. **ACM Transactions on Database Systems (TODS)**, v. 10, n. 1, p. 1–28, 1985. Available in: https://doi.org/10.1145/3148.3160. Access in: Tue, 06 Nov 2018 12:51:47 +0100.

FREEDMAN, C.; ISMERT, E.; LARSON, P. Compilation in the microsoft SQL server hekaton engine. **IEEE Database Engineering Bulletin**, v. 37, n. 1, p. 22–30, 2014. Available in: http://sites.computer.org/debull/A14mar/p22.pdf. Access in: Tue, 10 Mar 2020 16:23:50 +0100.

FUNKE, F.; KEMPER, A.; MÜHLBAUER, T.; NEUMANN, T.; LEIS, V. Hyper beyond software: Exploiting modern hardware for main-memory database systems. **Datenbank-Spektrum**, v. 14, n. 3, p. 173–181, 2014. Available in: https://doi.org/10.1007/s13222-014-0165-y. Access in: Mon, 10 Aug 2020 09:10:42 +0200.

GAO, S.; XU, J.; HE, B.; CHOI, B.; HU, H. Pcmlogging: reducing transaction logging overhead with PCM. In: MACDONALD, C.; OUNIS, I.; RUTHVEN, I. (Ed.). **Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011**. ACM, 2011. p. 2401–2404. Available in: https://doi.org/10.1145/2063576.2063977. Access in: Mon, 23 Dec 2019 15:05:33 +0100.

GARCIA-MOLINA, H.; SALEM, K. Main memory database systems: An overview. **IEEE Transactions on Knowledge and Data**, v. 4, n. 6, p. 509–516, 1992. Available in: https://doi.org/10.1109/69.180602. Access in: Sat, 20 May 2017 00:24:23 +0200.

GARG, V.; SINGH, A.; HARITSA, J. R. **On improving write performance in PCM databases**. [*S. l.*], 2015. Available in: https://dsl.cds.iisc.ac.in/publications/report/TR/TR-2015-01.pdf.

Tandem database group - nonstop SQL: A distributed, high-performance, high-availability implementation of SQL. In: GAWLICK, D.; HAYNIE, M. N.; REUTER, A. (Ed.). **High Performance Transaction Systems, 2nd International Workshop, Asilomar Conference Center, Pacific Grove, California, USA, September 28-30, 1987, Proceedings**. Springer, 1987. (Lecture Notes in Computer Science, v. 359), p. 60–104. Available in: https://doi.org/10.1007/3-540-51085-0_43. Access in: Tue, 14 May 2019 10:00:50 +0200.

GAWLICK, D.; KINKADE, D. Varieties of concurrency control in IMS/VS fast path. **IEEE Database Eng. Bull.**, v. 8, n. 2, p. 3–10, 1985. Available in: http://sites.computer.org/debull/85JUN-CD.pdf. Access in: Tue, 10 Mar 2020 16:23:50 +0100.

GOMES, D. B.; BRAYNER, A.; MACHADO, J. C. ETERNAL: uma estratégia eficiente de tolerância a falhas utilizando memória não volátil. In: **XXXIV Simpósio Brasileiro de Banco de Dados, SBBD 2019, Fortaleza, CE, Brazil, October 7-10, 2019**. SBC, 2019. p. 97–108. Available in: https://doi.org/10.5753/sbbd.2019.8811. Access in: Wed, 12 Feb 2020 18:21:06 +0100.

GRAEFE, G. Sorting and indexing with partitioned b-trees. In: **First Biennial Conference on Innovative Data Systems Research, CIDR 2003, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings**. www.cidrdb.org, 2003. Available in: http://www-db.cs.wisc.edu/cidr/cidr2003/program/p1.pdf. Access in: Tue, 23 Mar 2021 15:38:30 +0100.

GRAEFE, G.; GUY, W.; SAUER, C. **Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover, Second Edition**. Morgan & Claypool Publishers, 2016. (Synthesis Lectures on Data Management). Available in: https://doi.org/10.2200/S00710ED2V01Y201603DTM044. Access in: Tue, 16 May 2017 14:24:20 +0200.

GRAEFE, G.; KUNO, H. A. Definition, detection, and recovery of single-page failures, a fourth class of database failures. **Proceedings of the VLDB Endowment**, v. 5, n. 7, p. 646–655, 2012. Available in: http://vldb.org/pvldb/vol5/p646_goetzgraefe_vldb2012.pdf. Access in: Sat, 25 Apr 2020 13:59:05 +0200.

GRAEFE, G.; MCKENNA, W. J. The volcano optimizer generator: Extensibility and efficient search. In: **Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria**. IEEE Computer Society, 1993. p. 209–218. Available in: https://doi.org/10.1109/ICDE.1993.344061. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

GRAEFE, G.; SAUER, C.; GUY, W.; HÄRDER, T. Instant recovery with write-ahead logging. **Datenbank-Spektrum**, v. 15, n. 3, p. 235–239, 2015. Available in: https://doi.org/10.1007/s13222-015-0204-3. Access in: Thu, 18 May 2017 09:53:50 +0200.

GRAY, J.; HELLAND, P.; O'NEIL, P. E.; SHASHA, D. E. The dangers of replication and a solution. ACM Press, p. 173–182, 1996. Available in: https://doi.org/10.1145/233269.233330. Access in: Mon, 21 Jun 2021 16:00:48 +0200.

GRAY, J.; LORIE, R. A.; PUTZOLU, G. R.; TRAIGER, I. L. Granularity of locks in a large shared data base. In: KERR, D. S. (Ed.). **Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA**. ACM, 1975. p. 428–451. Available in: https://doi.org/10.1145/1282480.1282513. Access in: Tue, 06 Nov 2018 16:57:17 +0100.

GRAY, J.; MCJONES, P. R.; BLASGEN, M. W.; LINDSAY, B. G.; LORIE, R. A.; PRICE, T. G.; PUTZOLU, G. R.; TRAIGER, I. L. The recovery manager of the system R database manager. **ACM Computing Surveys (CSUR)**, v. 13, n. 2, p. 223–243, 1981. Available in: https://doi.org/10.1145/356842.356847. Access in: Tue, 06 Nov 2018 12:50:48 +0100.

GRAY, J.; REUTER, A. **Transaction Processing: Concepts and Techniques**. [*S. l.*]: Morgan Kaufmann, 1993. ISBN 1-55860-190-2. Access in: Mon, 06 Nov 2017 16:35:10 +0100.

GRUENWALD, L.; EICH, M. H. MMDB reload algorithms. In: CLIFFORD, J.; KING, R. (Ed.). **Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, USA, May 29-31, 1991**. ACM Press, 1991. p. 397–405. Available in: https://doi.org/10.1145/115790.115858. Access in: Tue, 06 Nov 2018 11:07:39 +0100.

GRUENWALD, L.; EICH, M. H. MMDB reload concerns. **Information sciences**, v. 76, n. 1-2, p. 151–176, 1994. Available in: https://doi.org/10.1016/0020-0255(94)90073-6. Access in: Sat, 27 May 2017 14:24:47 +0200.

GRUENWALD, L.; HUANG, J.; DUNHAM MARGARET H ANA MODEL OF CRASH RECOVERY IN MAIN MEMORY DATABASED LIN, J.-L.; PELTIER, A. C. Recovery in main memory databases. Citeseer, 1996.

Guo, W.; Hu, Z. Memory database index optimization. In: **2010 International Conference on Computational Intelligence and Software Engineering**. [*S. n.*], 2010. p. 1–3. Available in: https://doi.org/10.1109/CISE.2010.5676836. Access in: July 22, 2020.

GUPTA, M. K.; VERMA, V.; VERMA, M. S. In-memory database systems - A paradigm shift. **CoRR**, abs/1402.1258, 2014. Available in: http://arxiv.org/abs/1402.1258. Access in: Mon, 13 Aug 2018 16:46:56 +0200.

HAGMANN, R. B. Crash recovery scheme for a memory-resident database system. **IEEE Computer Architecture Letters**, v. 35, n. 9, p. 839–843, 1986. Available in: https://doi.org/10.1109/TC.1986.1676845. Access in: Sat, 20 May 2017 00:24:38 +0200.

HAGMANN, R. B. Reimplementing the cedar file system using logging and group commit. In: BELADY, L. (Ed.). **Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987**. ACM, 1987. p. 155–162. Available in: https://doi.org/10.1145/41457.37518. Access in: Tue, 06 Nov 2018 16:59:32 +0100.

HÄRDER, T.; REUTER, A. Principles of transaction-oriented database recovery. **ACM Computing Surveys (CSUR)**, v. 15, n. 4, p. 287–317, 1983. Available in: https://doi.org/10.1145/289.291. Access in: Wed, 14 Nov 2018 10:31:46 +0100.

HARIZOPOULOS, S.; ABADI, D. J.; MADDEN, S.; STONEBRAKER, M. OLTP through the looking glass, and what we found there. In: BRODIE, M. L. (Ed.). **Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker**. ACM / Morgan & Claypool, 2019. p. 409–439. Available in: https://doi.org/10.1145/3226595.3226635. Access in: Thu, 28 Mar 2019 09:33:20 +0100.

HARRIS, R. **Windows leaps into the NVM revolution**. 2016. Accessed in July 22, 2020. Available in: https://www.zdnet.com/article/windows-leaps-into-the-nvm-revolution/. Access in: July 22, 2020.

HAZENBERG, W.; HEMMINGA, S. Main memory database systems: Opportunities and pitfalls. **SC@ RUG 2011 proceedings**, p. 113, 2011.

HELLERSTEIN, J. M.; STONEBRAKER, M.; HAMILTON, J. R. **Architecture of a Database System**. [*S. n.*], 2007. v. 1. 141–259 p. Available in: https://doi.org/10.1561/1900000002. Access in: Sat, 25 Apr 2020 14:00:55 +0200.

HERLIHY, M.; MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In: SMITH, A. J. (Ed.). **Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993**. ACM, 1993. p. 289–300. Available in: https://doi.org/10.1145/165123.165164. Access in: Wed, 16 Oct 2019 14:14:49 +0200.

HOLANDA, M.; BRAYNER, A.; FIALHO, S. A self-adaptable scheduler for synchronizing transactions in dynamically configurable environments. **Data & Knowledge Engineering**, v. 66, n. 2, p. 223–242, 2008. Available in: https://doi.org/10.1016/j.datak.2008.02.004. Access in: Fri, 27 Dec 2019 21:14:08 +0100.

HU, H.; ZHOU, X.; ZHU, T.; QIAN, W.; ZHOU, A. In-memory transaction processing: efficiency and scalability considerations. **Knowledge and Information Systems**, v. 61, n. 3, p. 1209–1240, 2019. Available in: https://doi.org/10.1007/s10115-019-01340-7. Access in: Wed, 30 Oct 2019 08:36:16 +0100.

HUANG, J.; SCHWAN, K.; QURESHI, M. K. Nvram-aware logging in transaction systems. **Proceedings of the VLDB Endowment**, v. 8, n. 4, p. 389–400, 2014. Available in: http://www.vldb.org/pvldb/vol8/p389-huang.pdf. Access in: Sat, 25 Apr 2020 13:59:09 +0200.

HVASSHOVD, S.; TORBJØRNSEN, Ø.; BRATSBERG, S. E.; HOLAGER, P. The clustra telecom database: High availability, high throughput, and real-time response. In: DAYAL, U.; GRAY, P. M. D.; NISHIO, S. (Ed.). **VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland**. Morgan Kaufmann, 1995. p. 469–477. Available in: http://www.vldb.org/conf/1995/P469.PDF. Access in: Tue, 20 Feb 2018 15:19:44 +0100.

IBRAHIM, A. A. Z. A.; VARRETTE, S.; BOUVRY, P. PRESENCE: toward a novel approach for performance evaluation of mobile cloud saas web services. In: **2018 International Conference on Information Networking, ICOIN 2018, Chiang Mai, Thailand, January 10-12, 2018**. IEEE, 2018. p. 50–55. Available in: https://doi.org/10.1109/ICOIN.2018.8343082. Access in: Sun, 25 Jul 2021 11:49:19 +0200.

IZRAELEVITZ, J.; KELLY, T.; KOLLI, A. Failure-atomic persistent memory updates via JUSTDO logging. In: CONTE, T.; ZHOU, Y. (Ed.). **Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016**. ACM, 2016. p. 427–442. Available in: https://doi.org/10.1145/2872362.2872410. Access in: Tue, 03 Sep 2019 08:30:33 +0200.

JAGADISH, H. V.; LIEUWEN, D. F.; RASTOGI, R.; SILBERSCHATZ, A.; SUDARSHAN, S. Dalí: A high performance main memory storage manager. In: BOCCA, J. B.; JARKE, M.; ZANIOLO, C. (Ed.). **VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile**. Morgan Kaufmann, 1994. p. 48–59. Available in: http://www.vldb.org/conf/1994/P048.PDF. Access in: Wed, 29 Mar 2017 16:45:24 +0200.

JAGADISH, H. V.; SILBERSCHATZ, A.; SUDARSHAN, S. Recovering from main-memory lapses. In: AGRAWAL, R.; BAKER, S.; BELL, D. A. (Ed.). **19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings**. Morgan Kaufmann, 1993. p. 391–404. Available in: http://www.vldb.org/conf/1993/P391.PDF. Access in: Tue, 07 Nov 2017 16:24:37 +0100.

JOHNSON, R.; PANDIS, I.; HARDAVELLAS, N.; AILAMAKI, A.; FALSAFI, B. Shore-mt: a scalable storage manager for the multicore era. In: KERSTEN, M. L.; NOVIKOV, B.; TEUBNER, J.; POLUTIN, V.; MANEGOLD, S. (Ed.). **EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings**. ACM, 2009. (ACM International Conference Proceeding Series, v. 360), p. 24–35. Available in: https://doi.org/10.1145/1516360.1516365. Access in: Thu, 02 Dec 2021 17:30:55 +0100.

JUNQUEIRA, F. P.; REED, B. C.; SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In: **Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011**. IEEE Compute Society, 2011. p. 245–256. Available in: https://doi.org/10.1109/DSN.2011.5958223. Access in: Sun, 25 Oct 2020 22:57:32 +0100.

KALLMAN, R.; KIMURA, H.; NATKINS, J.; PAVLO, A.; RASIN, A.; ZDONIK, S. B.; JONES, E. P. C.; MADDEN, S.; STONEBRAKER, M.; ZHANG, Y.; HUGG, J.; ABADI, D. J. H-store: a high-performance, distributed main memory transaction processing system. **Proceedings of the VLDB Endowment**, v. 1, n. 2, p. 1496–1499, 2008. Available in: http://www.vldb.org/pvldb/vol1/1454211.pdf. Access in: Sat, 25 Apr 2020 13:59:42 +0200.

KARNAGEL, T.; DEMENTIEV, R.; RAJWAR, R.; LAI, K.; LEGLER, T.; SCHLEGEL, B.; LEHNER, W. Improving in-memory database index performance with intel® transactional synchronization extensions. In: **20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014**. IEEE Computer Society, 2014. p. 476–487. Available in: https://doi.org/10.1109/HPCA.2014.6835957. Access in: Wed, 16 Oct 2019 14:14:50 +0200.

KEMPER, A.; NEUMANN, T. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In: ABITEBOUL, S.; BÖHM, K.; KOCH, C.; TAN, K. (Ed.). **Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany**. IEEE Computer Society, 2011. p. 195–206. Available in: https://doi.org/10.1109/ICDE.2011.5767867. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

KIM, C.; CHHUGANI, J.; SATISH, N.; SEDLAR, E.; NGUYEN, A. D.; KALDEWEY, T.; LEE, V. W.; BRANDT, S. A.; DUBEY, P. FAST: fast architecture sensitive tree search on modern cpus and gpus. In: ELMAGARMID, A. K.; AGRAWAL, D. (Ed.). **Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010**. ACM, 2010. p. 339–350. Available in: https://doi.org/10.1145/1807167.1807206. Access in: Tue, 06 Nov 2018 11:07:38 +0100.

KIM, J.-J.; KANG, J.-J.; LEE, K.-Y. Recovery methods in main memory dbms. **International journal of advanced smart convergence**, The institute Of Webcasting, Internet And Telecommunication, v. 1, n. 2, p. 26–29, 2012.

KIM, L.; LEE, E. ROVN: replica placement for distributed data system with heterogeneous memory devices. **IEICE Electron. Express**, v. 18, n. 23, p. 20210379, 2021. Available in: https://doi.org/10.1587/elex.18.20210379. Access in: Thu, 03 Feb 2022 13:12:45 +0100.

KIMURA, H. FOEDUS: OLTP engine for a thousand cores and NVRAM. In: SELLIS, T. K.; DAVIDSON, S. B.; IVES, Z. G. (Ed.). **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015**. ACM, 2015. p. 691–706. Available in: https://doi.org/10.1145/2723372.2746480. Access in: Tue, 06 Nov 2018 11:07:39 +0100.

KUNG, H. T.; ROBINSON, J. T. On optimistic methods for concurrency control. **ACM Transactions on Database Systems (TODS)**, v. 6, n. 2, p. 213–226, 1981. Available in: https://doi.org/10.1145/319566.319567. Access in: Tue, 28 Apr 2020 13:45:05 +0200.

LAHIRI, T.; NEIMAT, M.; FOLKMAN, S. Oracle timesten: An in-memory database for enterprise applications. **IEEE Database Engineering Bulletin**, v. 36, n. 2, p. 6–13, 2013. Available in: http://sites.computer.org/debull/A13june/TimesTen1.pdf. Access in: Tue, 10 Mar 2020 16:23:50 +0100.

LAMPORT, L. Paxos made simple. **ACM Sigact News**, v. 32, n. 4, p. 18–25, 2001.

LAMPORT, L. The part-time parliament. In: MALKHI, D. (Ed.). **Concurrency: the Works of Leslie Lamport**. ACM, 2019. p. 277–317. Available in: https://doi.org/10.1145/3335772.3335939. Access in: Tue, 07 Apr 2020 17:11:02 +0200.

LARSON, P. Linear hashing with separators - A dynamic hashing scheme achieving one-access retrieval. **ACM Transactions on Database Systems (TODS)**, v. 13, n. 3, p. 366–388, 1988. Available in: https://doi.org/10.1145/44498.44500. Access in: Tue, 06 Nov 2018 12:51:47 +0100.

LARSON, P.; LEVANDOSKI, J. J. Modern main-memory database systems. **Proceedings of the VLDB Endowment**, v. 9, n. 13, p. 1609–1610, 2016. Available in: http://www.vldb.org/pvldb/vol9/p1609-larson.pdf. Access in: Sat, 25 Apr 2020 13:59:12 +0200.

LARSON, P.; ZWILLING, M.; FARLEE, K. The hekaton memory-optimized OLTP engine. **IEEE Database Engineering Bulletin**, v. 36, n. 2, p. 34–40, 2013. Available in: http://sites.computer.org/debull/A13june/Hekaton1.pdf. Access in: Tue, 10 Mar 2020 16:23:50 +0100.

LEE, J.; KWON, Y. S.; FÄRBER, F.; MUEHLE, M.; LEE, C.; BENSBERG, C.; LEE, J.; LEE, A. H.; LEHNER, W. SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In: JENSEN, C. S.; JERMAINE, C. M.; ZHOU, X. (Ed.). **29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013**. IEEE Computer Society, 2013. p. 1165–1173. Available in: https://doi.org/10.1109/ICDE.2013.6544906. Access in: Thu, 21 Nov 2019 11:11:44 +0100.

LEHMAN, T. J.; CAREY, M. J. A study of index structures for main memory database management systems. In: CHU, W. W.; GARDARIN, G.; OHSUGA, S.; KAMBAYASHI, Y. (Ed.). **VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings**. Morgan Kaufmann, 1986. p. 294–303. Available in: http://www.vldb.org/conf/1986/P294.PDF. Access in: Tue, 07 Nov 2017 06:51:45 +0100.

LEHMAN, T. J.; CAREY, M. J. A recovery algorithm for A high-performance memory-resident database system. In: DAYAL, U.; TRAIGER, I. L. (Ed.). **Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987**. ACM Press, 1987. p. 104–117. Available in: https://doi.org/10.1145/38713.38730. Access in: Tue, 06 Nov 2018 11:07:37 +0100.

LEHMAN, T. J.; SHEKITA, E. J.; CABRERA, L. An evaluation of starburst's memory resident storage component. **IEEE Transactions on Knowledge and Data**, v. 4, n. 6, p. 555–566, 1992. Available in: https://doi.org/10.1109/69.180606. Access in: Sat, 20 May 2017 00:24:20 +0200.

LEIS, V.; BONCZ, P. A.; KEMPER, A.; NEUMANN, T. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In: DYRESON, C. E.; LI, F.; ÖZSU, M. T. (Ed.). **International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014**. ACM, 2014. p. 743–754. Available in: https://doi.org/10.1145/2588555.2610507. Access in: Tue, 06 Nov 2018 11:07:39 +0100.

LEIS, V.; HAUBENSCHILD, M.; NEUMANN, T. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. **IEEE Database Engineering Bulletin**, v. 42, n. 1, p. 73–84, 2019. Available in: http://sites.computer.org/debull/A19mar/p73.pdf. Access in: Tue, 10 Mar 2020 16:23:50 +0100.

LEIS, V.; KEMPER, A.; NEUMANN, T. The adaptive radix tree: Artful indexing for main-memory databases. In: JENSEN, C. S.; JERMAINE, C. M.; ZHOU, X. (Ed.). **29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013**. IEEE Computer Society, 2013. p. 38–49. Available in: https://doi.org/10.1109/ICDE.2013.6544812. Access in: Thu, 21 Nov 2019 11:11:44 +0100.

LEIS, V.; KEMPER, A.; NEUMANN, T. Exploiting hardware transactional memory in main-memory databases. In: CRUZ, I. F.; FERRARI, E.; TAO, Y.; BERTINO, E.; TRAJCEVSKI, G. (Ed.). **IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014**. IEEE Computer Society, 2014. p. 580–591. Available in: https://doi.org/10.1109/ICDE.2014.6816683. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

LEVANDOSKI, J. J.; LARSON, P.; STOICA, R. Identifying hot and cold data in main-memory databases. In: JENSEN, C. S.; JERMAINE, C. M.; ZHOU, X. (Ed.). **29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013**. IEEE Computer Society, 2013. p. 26–37. Available in: https://doi.org/10.1109/ICDE.2013.6544811. Access in: Thu, 21 Nov 2019 11:11:44 +0100.

LEVANDOSKI, J. J.; LOMET, D. B.; SENGUPTA, S. The bw-tree: A b-tree for new hardware platforms. In: JENSEN, C. S.; JERMAINE, C. M.; ZHOU, X. (Ed.). **29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013**. IEEE Computer Society, 2013. p. 302–313. Available in: https://doi.org/10.1109/ICDE.2013.6544834. Access in: Thu, 21 Nov 2019 11:11:44 +0100.

LEVY, E.; SILBERSCHATZ, A. Incremental recovery in main memory database systems. **IEEE Transactions on Knowledge and Data**, v. 4, n. 6, p. 529–540, 1992. Available in: https://doi.org/10.1109/69.180604. Access in: Sat, 20 May 2017 00:24:23 +0200.

LI, K.; NAUGHTON, J. F. Multiprocessor main memory transaction processing. In: JAJODIA, S.; KIM, W.; SILBERSCHATZ, A. (Ed.). **Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, Austin, Texas, USA, December 5-7, 1988**. IEEE Computer Society, 1988. p. 177–187. Available in: https://doi.org/10.1109/DPDS.1988.675014. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

LI, L.; WANG, G.; WU, G.; YUAN, Y.; CHEN, L.; LIAN, X. A comparative study of consistent snapshot algorithms for main-memory database systems. **CoRR**, abs/1810.04915, 2018. Available in: http://arxiv.org/abs/1810.04915. Access in: Thu, 08 Oct 2020 16:48:33 +0200.

LI, L.; WANG, G.; WU, G.; YUAN, Y. Consistent snapshot algorithms for in-memory database systems: Experiments and analysis. In: **34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018**. IEEE Computer Society, 2018. p. 1284–1287. Available in: https://doi.org/10.1109/ICDE.2018.00131. Access in: Thu, 08 Oct 2020 16:48:34 +0200.

LI, X.; EICH, M. H. Post-crash log processing for fuzzy checkpointing main memory databases. In: **Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria**. IEEE Computer Society, 1993. p. 117–124. Available in: https://doi.org/10.1109/ICDE.1993.344071. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

LI, Y.; PANDIS, I.; MÜLLER, R.; RAMAN, V.; LOHMAN, G. M. Numa-aware algorithms: the case of data shuffling. In: **CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings**. www.cidrdb.org, 2013. Available in: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper121.pdf. Access in: Wed, 30 Sep 2020 19:15:08 +0200.

LIEDES, A.; WOLSKI, A. SIREN: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases. In: LIU, L.; REUTER, A.; WHANG, K.; ZHANG, J. (Ed.). **Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA**. IEEE Computer Society, 2006. p. 99. Available in: https://doi.org/10.1109/ICDE.2006.140. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

LIN, J.; DUNHAM, M. H.; NASCIMENTO, M. A. A survey of distributed database checkpointing. **Distributed Parallel Databases**, v. 5, n. 3, p. 289–319, 1997. Available in: https://doi.org/10.1023/A:1008689312900. Access in: Mon, 18 May 2020 12:42:53 +0200.

LISKOV, B.; SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, v. 5, n. 3, p. 381–404, 1983. Available in: https://doi.org/10.1145/2166.357215. Access in: Tue, 06 Nov 2018 12:51:29 +0100.

LITWIN, W. Linear hashing: A new tool for file and table addressing. In: **Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings**. IEEE Computer Society, 1980. p. 212–223. Available in: https://www.cs.bu.edu/faculty/gkollios/ada17/LectNotes/linear-hashing.PDF. Access in: Wed, 29 Mar 2017 16:45:24 +0200.

LO, J. L.; BARROSO, L. A.; EGGERS, S. J.; GHARACHORLOO, K.; LEVY, H. M.; PAREKH, S. S. An analysis of database workload performance on simultaneous multithreaded processors. In: VALERO, M.; SOHI, G. S.; DEGROOT, D. (Ed.). **Proceedings of the 25th**

Annual International Symposium on Computer Architecture, ISCA 1998, Barcelona, Spain, June 27 - July 1, 1998. IEEE Computer Society, 1998. p. 39–50. Available in: https://doi.org/10.1109/ISCA.1998.694761. Access in: Wed, 16 Oct 2019 14:14:49 +0200.

LOMET, D. B.; FEKETE, A. D.; WANG, R.; WARD, P. Multi-version concurrency via timestamp range conflict management. In: KEMENTSIETSIDIS, A.; SALLES, M. A. V. (Ed.). **IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012**. IEEE Computer Society, 2012. p. 714–725. Available in: https://doi.org/10.1109/ICDE.2012.10. Access in: Thu, 27 Aug 2020 11:23:11 +0200.

MAAS, L. M.; KISSINGER, T.; HABICH, D.; LEHNER, W. BUZZARD: a numa-aware in-memory indexing system. In: ROSS, K. A.; SRIVASTAVA, D.; PAPADIAS, D. (Ed.). **Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013**. ACM, 2013. p. 1285–1286. Available in: https://doi.org/10.1145/2463676.2465342. Access in: Tue, 06 Nov 2018 11:07:37 +0100.

MADRIA, S. K. A study of the concurrency control and recovery algorithms in nested transaction environment. **Comput. J.**, v. 40, n. 10, p. 630–639, 1997. Available in: https://doi.org/10.1093/comjnl/40.10.630. Access in: Sat, 20 May 2017 00:22:25 +0200.

MAGALHÃES, A. Main memory databases instant recovery. In: BERNSTEIN, P. A.; RABL, T. (Ed.). **Proceedings of the VLDB 2021 PhD Workshop co-located with the 47th International Conference on Very Large Databases (VLDB 2021), Copenhagen, Denmark, August 16, 2021**. CEUR-WS.org, 2021. (CEUR Workshop Proceedings, v. 2971). Available in: http://ceur-ws.org/Vol-2971/paper10.pdf. Access in: Mon, 15 Nov 2021 14:58:09 +0100.

MAGALHÃES, A.; BRAYNER, A.; MONTEIRO, J. M.; MORAES, G. Indexed log file: Towards main memory database instant recovery. In: VELEGRAKIS, Y.; ZEINALIPOUR-YAZTI, D.; CHRYSANTHIS, P. K.; GUERRA, F. (Ed.). **Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021**. OpenProceedings.org, 2021. p. 355–360. Available in: https://doi.org/10.5441/002/edbt.2021.34. Access in: Sat, 20 Mar 2021 10:29:38 +0100.

MAGALHÃES, A.; MONTEIRO, J. M.; BRAYNER, A. Ajuste de performance em bancos de dados nosql. In: **III Escola Reginal de Informática do Piauí , ERIPI 2017, Picos, PI, Brazil, 2017**. [*S. l.*: *s. n.*], 2017.

MAGALHÃES, A.; MONTEIRO, J. M.; BRAYNER, A. Gerenciamento e processamento de big data com bancos de dados em memória. In: **I Jornada latino-americana de atualização em informática , JOLAI 2017, São Paulo, SP, Brazil, 2018**. [*S. n.*], 2018. Available in: https://sol.sbc.org.br/index.php/jolai_clei/issue/view/283/jolai_clei. Access in: July 22, 2020.

MAGALHÃES, A.; MONTEIRO, J. M.; BRAYNER, A. Sistemas de gerenciamento de banco de dados em memória. In: **XIV Simpósio Brasileiro de Sistemas de Informação , SBSI 2018, Caxias do Sul, RS, Brazil, 2018**. [*S. n.*], 2018. Available in: https://www.ucs.br/site/midia/arquivos/topicos-sistema-informacao.pdf. Access in: July 22, 2020.

MAGALHÃES, A.; MONTEIRO, J. M.; BRAYNER, A. Main memory databases instant recovery. In: **Database Theses and Dissertations Workshop / 34th Brazilian Symposium on**

Databases , SBBD 2019, Fortaleza, CE, Brazil, October 7-10, 2019. [*S. n.*], 2019. Available in: https://sbbd.org.br/2019/wp-content/uploads/sites/6/2020/01/Procedings-Companion.pdf. Access in: Mon, 13 Sep 2021 19:43:48 +0200.

MAGALHÃES, A.; MONTEIRO, J. M.; BRAYNER, A. Main memory database recovery: A survey. **ACM Comput. Surv.**, v. 54, n. 2, p. 46:1–46:36, 2021. Available in: https://doi.org/10.1145/3442197. Access in: Wed, 19 May 2021 16:17:39 +0200.

MAKRESHANSKI, D.; LEVANDOSKI, J. J.; STUTSMAN, R. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. **Proceedings of the VLDB Endowment**, v. 8, n. 11, p. 1298–1309, 2015. Available in: http://www.vldb.org/pvldb/vol8/p1298-makreshanski.pdf. Access in: Sat, 25 Apr 2020 13:58:37 +0200.

MALVIYA, N.; WEISBERG, A.; MADDEN, S.; STONEBRAKER, M. Rethinking main memory OLTP recovery. In: CRUZ, I. F.; FERRARI, E.; TAO, Y.; BERTINO, E.; TRAJCEVSKI, G. (Ed.). **IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014**. IEEE Computer Society, 2014. p. 604–615. Available in: https://doi.org/10.1109/ICDE.2014.6816685. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

MAO, Y.; KOHLER, E.; MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In: FELBER, P.; BELLOSA, F.; BOS, H. (Ed.). **European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012**. ACM, 2012. p. 183–196. Available in: https://doi.org/10.1145/2168836.2168855. Access in: Tue, 06 Nov 2018 16:58:31 +0100.

MEMTIER BENCHMARK. **GitHub - RedisLabs/memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool**. 2020. Available in: https://github.com/RedisLabs/memtier_benchmark. Access in: August 26, 2020.

MENON, P.; PAVLO, A.; MOWRY, T. C. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. **Proceedings of the VLDB Endowment**, v. 11, n. 1, p. 1–13, 2017. Available in: http://www.vldb.org/pvldb/vol11/p1-menon.pdf. Access in: Sat, 25 Apr 2020 13:59:08 +0200.

MINOURA, T. Multi-level concurrency control of a database system. In: **Fourth Symposium on Reliability in Distributed Software and Database Systems, SRDS 1984, Silver Spring, Maryland, USA, October 15-17, 1984, Proceedings**. [*S. l.*]: IEEE Computer Society, 1984. p. 156–168. Access in: Fri, 09 Jan 2015 14:54:29 +0100.

MITCHELL, C.; GENG, Y.; LI, J. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In: BIRRELL, A.; SIRER, E. G. (Ed.). **2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013**. USENIX Association, 2013. p. 103–114. Available in: https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell. Access in: Mon, 01 Feb 2021 08:43:29 +0100.

MOHAN, C. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In: MCLEOD, D.; SACKS-DAVIS, R.; SCHEK, H. (Ed.). **16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings**. Morgan Kaufmann, 1990. p. 392–405. Available in: http://www.vldb.org/conf/1990/P392.PDF. Access in: Wed, 29 Mar 2017 16:45:23 +0200.

MOHAN, C. ARIES/LHS: A concurrency control and recovery method using write-ahead logging for linear hashing with separators. In: **Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria**. IEEE Computer Society, 1993. p. 243–252. Available in: https://doi.org/10.1109/ICDE.1993.344058. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

MOHAN, C. Repeating history beyond ARIES. In: ATKINSON, M. P.; ORLOWSKA, M. E.; VALDURIEZ, P.; ZDONIK, S. B.; BRODIE, M. L. (Ed.). **VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK**. Morgan Kaufmann, 1999. p. 1–17. Available in: http://www.vldb.org/conf/1999/P1.pdf. Access in: Thu, 12 Mar 2020 11:33:39 +0100.

MOHAN, C.; HADERLE, D.; LINDSAY, B. G.; PIRAHESH, H.; SCHWARZ, P. M. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. **ACM Transactions on Database Systems (TODS)**, v. 17, n. 1, p. 94–162, 1992. Available in: https://doi.org/10.1145/128765.128770. Access in: Wed, 29 May 2019 10:39:45 +0200.

MOHAN, C.; LEVINE, F. E. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In: STONEBRAKER, M. (Ed.). **Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992**. ACM Press, 1992. p. 371–380. Available in: https://doi.org/10.1145/130283.130338. Access in: Tue, 06 Nov 2018 11:07:39 +0100.

MOHAN, C.; NARANG, I. ARIES/CSA: A method for database recovery in client-server architectures. In: SNODGRASS, R. T.; WINSLETT, M. (Ed.). **Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994**. ACM Press, 1994. p. 55–66. Available in: https://doi.org/10.1145/191839.191849. Access in: Tue, 06 Nov 2018 11:07:37 +0100.

MOHAN, C.; PIRAHESH, H. ARIES-RRH: restricted repeating of history in the ARIES transaction recovery method. In: **Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan**. IEEE Computer Society, 1991. p. 718–727. Available in: https://doi.org/10.1109/ICDE.1991.131521. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

MONTEIRO, J. M.; BRAYNER, A.; LIFSCHITZ, S. Using transaction isolation levels for ensuring replicated database consistency in mobile computing environments. In: KOTIDIS, Y.; MARRÓN, P. J.; GRUENWALD, L.; ZEINALIPOUR-YAZTI, D. (Ed.). **Eighth ACM International Workshop on Data Engineering for Wireless and Mobile Access, Mobide 2009, June 29, 2009, Providence, Rhode Island, USA, Proceedings**. ACM, 2009. p. 1–8. Available in: https://doi.org/10.1145/1594139.1594146. Access in: Tue, 06 Nov 2018 16:58:57 +0100.

MÜHE, H.; KEMPER, A.; NEUMANN, T. How to efficiently snapshot transactional data: hardware or software controlled? In: HARIZOPOULOS, S.; LUO, Q. (Ed.). **Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, June 13, 2011**. ACM, 2011. p. 17–26. Available in: https://doi.org/10.1145/1995441.1995444. Access in: Tue, 06 Nov 2018 16:58:57 +0100.

MÜHLBAUER, T.; RÖDIGER, W.; SEILBECK, R.; REISER, A.; KEMPER, A.; NEUMANN, T. Instant loading for main memory databases. **Proceedings of the VLDB Endowment**, v. 6, n. 14, p. 1702–1713, 2013. Available in: http://www.vldb.org/pvldb/vol6/p1702-muehlbauer.pdf. Access in: Sat, 25 Apr 2020 13:59:09 +0200.

MYSQL. **MySQL**. 2020. Accessed in October 3, 2020. Available in: http://www.mysql.com. Access in: October 3, 2020.

NEUMANN, T. Efficiently compiling efficient query plans for modern hardware. **Proceedings of the VLDB Endowment**, v. 4, n. 9, p. 539–550, 2011. Available in: http://www.vldb.org/pvldb/vol4/p539-neumann.pdf. Access in: Sat, 25 Apr 2020 13:58:50 +0200.

NEUMANN, T.; MÜHLBAUER, T.; KEMPER, A. Fast serializable multi-version concurrency control for main-memory database systems. In: SELLIS, T. K.; DAVIDSON, S. B.; IVES, Z. G. (Ed.). **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015**. ACM, 2015. p. 677–689. Available in: https://doi.org/10.1145/2723372.2749436. Access in: Tue, 06 Nov 2018 11:07:38 +0100.

NORDBY, M.; KRZENSKI, S.; SAADI, F. A. Application load simulation and the potential for denial-of-service when the linux top program is misused. Available in: http://www.micsymposium.org/mics_2007/Nordby.pdf. Access in: December 12, 2020.

NYBERG, C.; BARCLAY, T.; CVETANOVIC, Z.; GRAY, J.; LOMET, D. B. Alphasort: A RISC machine sort. In: SNODGRASS, R. T.; WINSLETT, M. (Ed.). **Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994**. ACM Press, 1994. p. 233–242. Available in: https://doi.org/10.1145/191839.191884. Access in: Tue, 06 Nov 2018 11:07:38 +0100.

OH, J.-S.; LEE, S.-H. Database workload analysis: An empirical study. **The KIPS Transactions: PartD**, Korea Information Processing Society, v. 11, n. 4, p. 747–754, 2004.

OLSON, M. A.; BOSTIC, K.; SELTZER, M. I. Berkeley db. In: **USENIX Annual Technical Conference, FREENIX Track**. [*S. l.*: *s. n.*], 1999. p. 183–191.

O'NEIL, P. E.; CHENG, E.; GAWLICK, D.; O'NEIL, E. J. The log-structured merge-tree (lsm-tree). **Acta Informatica**, v. 33, n. 4, p. 351–385, 1996. Available in: https://doi.org/10.1007/s002360050048. Access in: Sun, 21 Jun 2020 17:38:20 +0200.

ONGARO, D.; OUSTERHOUT, J. K. In search of an understandable consensus algorithm. In: GIBSON, G.; ZELDOVICH, N. (Ed.). **2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014**. USENIX Association, 2014. p. 305–319. Available in: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro. Access in: Mon, 01 Feb 2021 08:43:54 +0100.

ONGARO, D.; RUMBLE, S. M.; STUTSMAN, R.; OUSTERHOUT, J. K.; ROSENBLUM, M. Fast crash recovery in ramcloud. In: WOBBER, T.; DRUSCHEL, P. (Ed.). **Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011**. ACM, 2011. p. 29–41. Available in: https://doi.org/10.1145/2043556.2043560. Access in: Tue, 06 Nov 2018 16:59:32 +0100.

ORACLE. **Oracle | Integrated Cloud Applications and Platform Services**. 2020. Accessed in December 4, 2020. Available in: http://www.oracle.com. Access in: December 4, 2020.

OUAKNINE, K.; AGRA, O.; GUZ, Z. Optimization of rocksdb for redis on flash. In: **Proceedings of the International Conference on Compute and Data Analysis, ICCDA 2017, Lakeland, FL, USA, May 19-23, 2017**. ACM, 2017. p. 155–161. Available in: https://doi.org/10.1145/3093241.3093278. Access in: Tue, 06 Nov 2018 11:07:48 +0100.

OUKID, I.; BOOSS, D.; LESPINASSE, A.; LEHNER, W.; WILLHALM, T.; GOMES, G. Memory management techniques for large-scale persistent-main-memory systems. **Proceedings of the VLDB Endowment**, v. 10, n. 11, p. 1166–1177, 2017. Available in: http://www.vldb.org/pvldb/vol10/p1166-oukid.pdf. Access in: Sat, 25 Apr 2020 13:58:45 +0200.

PALAKOLLU, S. M. Introduction to the linux environment. In: **Practical System Programming with C**. [*S. l.*]: Springer, 2021. p. 1–36.

PANDIS, I.; TÖZÜN, P.; JOHNSON, R.; AILAMAKI, A. PLP: page latch-free shared-everything OLTP. **Proceedings of the VLDB Endowment**, v. 4, n. 10, p. 610–621, 2011. Available in: http://www.vldb.org/pvldb/vol4/p610-pandis.pdf. Access in: Mon, 26 Oct 2020 08:50:11 +0100.

PAVLO, A.; ANGULO, G.; ARULRAJ, J.; LIN, H.; LIN, J.; MA, L.; MENON, P.; MOWRY, T. C.; PERRON, M.; QUAH, I.; SANTURKAR, S.; TOMASIC, A.; TOOR, S.; AKEN, D. V.; WANG, Z.; WU, Y.; XIAN, R.; ZHANG, T. Self-driving database management systems. In: **CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings**. www.cidrdb.org, 2017. Available in: http://cidrdb.org/cidr2017/papers/p42-pavlo-cidr17.pdf. Access in: Wed, 15 Jul 2020 07:56:53 +0200.

PELLEY, S.; WENISCH, T. F.; GOLD, B. T.; BRIDGE, B. Storage management in the NVRAM era. **Proceedings of the VLDB Endowment**, v. 7, n. 2, p. 121–132, 2013. Available in: http://www.vldb.org/pvldb/vol7/p121-pelley.pdf. Access in: Sat, 25 Apr 2020 13:59:21 +0200.

PELOTON. **Peloton - The Self-Driving Database Management System**. 2019. Accessed in April 30, 2020. Available in: https://pelotondb.io. Access in: April 30, 2020.

POROBIC, D.; LIAROU, E.; TÖZÜN, P.; AILAMAKI, A. Atrapos: Adaptive transaction processing on hardware islands. In: CRUZ, I. F.; FERRARI, E.; TAO, Y.; BERTINO, E.; TRAJCEVSKI, G. (Ed.). **IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014**. IEEE Computer Society, 2014. p. 688–699. Available in: https://doi.org/10.1109/ICDE.2014.6816692. Access in: Sun, 25 Oct 2020 23:05:27 +0100.

POROBIC, D.; PANDIS, I.; BRANCO, M.; TÖZÜN, P.; AILAMAKI, A. OLTP on hardware islands. **Proceedings of the VLDB Endowment**, v. 5, n. 11, p. 1447–1458, 2012. Available in: http://vldb.org/pvldb/vol5/p1447_danicaporobic_vldb2012.pdf. Access in: Mon, 26 Oct 2020 08:50:12 +0100.

PÖSS, M.; NAMBIAR, R. O.; WALRATH, D. Why you should run TPC-DS: A workload analysis. In: KOCH, C.; GEHRKE, J.; GAROFALAKIS, M. N.; SRIVASTAVA, D.; ABERER, K.; DESHPANDE, A.; FLORESCU, D.; CHAN, C. Y.; GANTI, V.; KANNE, C.; KLAS, W.; NEUHOLD, E. J. (Ed.). **Proceedings of the 33rd International Conference on Very**

Large Data Bases, University of Vienna, Austria, September 23-27, 2007. ACM, 2007. p. 1138–1149. Available in: http://www.vldb.org/conf/2007/papers/industrial/p1138-poess.pdf. Access in: Thu, 12 Mar 2020 11:33:41 +0100.

PUCHERAL, P.; THÉVENIN, J.; VALDURIEZ, P. Efficient main memory data management using the dbgraph storage model. In: MCLEOD, D.; SACKS-DAVIS, R.; SCHEK, H. (Ed.). **16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings**. Morgan Kaufmann, 1990. p. 683–695. Available in: http://www.vldb.org/conf/1990/P683.PDF. Access in: Wed, 29 Mar 2017 16:45:24 +0200.

RAMAKRISHNAN, R.; GEHRKE, J. **Database management systems (3. ed.)**. [*S. l.*]: McGraw-Hill, 2003. ISBN 978-0-07-115110-8. Access in: Thu, 14 Apr 2011 14:43:21 +0200.

RAMAN, V.; SWART, G.; QIAO, L.; REISS, F.; DIALANI, V.; KOSSMANN, D.; NARANG, I.; SIDLE, R. Constant-time query processing. In: ALONSO, G.; BLAKELEY, J. A.; CHEN, A. L. P. (Ed.). **Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico**. IEEE Computer Society, 2008. p. 60–69. Available in: https://doi.org/10.1109/ICDE.2008.4497414. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

RAO, J.; ROSS, K. A. Making b^+-trees cache conscious in main memory. In: CHEN, W.; NAUGHTON, J. F.; BERNSTEIN, P. A. (Ed.). **Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA**. ACM, 2000. p. 475–486. Available in: https://doi.org/10.1145/342009.335449. Access in: Tue, 06 Nov 2018 11:07:37 +0100.

RAOUX, S.; BURR, G. W.; BREITWISCH, M. J.; RETTNER, C. T.; CHEN, Y.; SHELBY, R. M.; SALINGA, M.; KREBS, D.; CHEN, S.; LUNG, H.; LAM, C. H. Phase-change random access memory: A scalable technology. **IBM Journal of Research and Development**, v. 52, n. 4-5, p. 465–480, 2008. Available in: https://doi.org/10.1147/rd.524.0465. Access in: Fri, 13 Mar 2020 10:55:07 +0100.

REDIS. **Redis**. 2020. Available in: https://redis.io. Access in: August 26, 2020.

REDIS LABS. **memtier_benchmark: A High-Throughput Benchmarking Tool for Redis & Memcached**. 2020. Available in: https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached. Access in: October 06, 2020.

REDIS LABS. **Redis Labs | The Best Redis Experience**. 2020. Available in: https://redislabs.com. Access in: October 06, 2020.

REN, K.; DIAMOND, T.; ABADI, D. J.; THOMSON, A. Low-overhead asynchronous checkpointing in main-memory database systems. In: ÖZCAN, F.; KOUTRIKA, G.; MADDEN, S. (Ed.). **Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016**. ACM, 2016. p. 1539–1551. Available in: https://doi.org/10.1145/2882903.2915966. Access in: Wed, 14 Nov 2018 10:56:20 +0100.

REN, K.; THOMSON, A.; ABADI, D. J. Lightweight locking for main memory database systems. **Proceedings of the VLDB Endowment**, v. 6, n. 2, p. 145–156, 2012. Available in: http://www.vldb.org/pvldb/vol6/p145-ren.pdf. Access in: Sat, 25 Apr 2020 13:59:13 +0200.

RENEN, A. van; LEIS, V.; KEMPER, A.; NEUMANN, T.; HASHIDA, T.; OE, K.; DOI, Y.; HARADA, L.; SATO, M. Managing non-volatile memory in database systems. In: DAS, G.; JERMAINE, C. M.; BERNSTEIN, P. A. (Ed.). **Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018**. ACM, 2018. p. 1541–1555. Available in: https://doi.org/10.1145/3183713.3196897. Access in: Sat, 19 Oct 2019 20:14:56 +0200.

RENEN, A. van; VOGEL, L.; LEIS, V.; NEUMANN, T.; KEMPER, A. Persistent memory I/O primitives. In: NEUMANN, T.; SALEM, K. (Ed.). **Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019**. ACM, 2019. p. 12:1–12:7. Available in: https://doi.org/10.1145/3329785.3329930. Access in: Sun, 25 Oct 2020 22:34:16 +0100.

RODEH, O. B-trees, shadowing, and clones. **ACM Transactions on Storage (TOS)**, v. 3, n. 4, p. 2:1–2:27, 2008. Available in: https://doi.org/10.1145/1326542.1326544. Access in: Thu, 30 Jul 2020 14:25:07 +0200.

ROSENBLUM, M.; BUGNION, E.; HERROD, S. A.; WITCHEL, E.; GUPTA, A. The impact of architectural trends on operating system performance. In: JONES, M. B. (Ed.). **Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995**. ACM, 1995. p. 285–298. Available in: https://doi.org/10.1145/224056.224078. Access in: Tue, 06 Nov 2018 16:59:32 +0100.

ROSENBLUM, M.; OUSTERHOUT, J. K. The design and implementation of a log-structured file system. **ACM Transactions on Computer Systems (TOCS)**, v. 10, n. 1, p. 26–52, 1992. Available in: https://doi.org/10.1145/146941.146943. Access in: Wed, 14 Nov 2018 10:49:57 +0100.

ROTHERMEL, K.; MOHAN, C. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In: APERS, P. M. G.; WIEDERHOLD, G. (Ed.). **Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands**. Morgan Kaufmann, 1989. p. 337–346. Available in: http://www.vldb.org/conf/1989/P337.PDF. Access in: Wed, 29 Mar 2017 16:45:23 +0200.

SAGIV, Y. Concurrent operations on b*-trees with overtaking. **Journal of computer and system sciences**, v. 33, n. 2, p. 275–296, 1986. Available in: https://doi.org/10.1016/0022-0000(86)90021-8. Access in: Sat, 20 May 2017 00:25:56 +0200.

SALEM, K.; GARCIA-MOLINA, H. Checkpointing memory-resident databases. In: **Proceedings of the Fifth International Conference on Data Engineering, February 6-10, 1989, Los Angeles, California, USA**. IEEE Computer Society, 1989. p. 452–462. Available in: https://doi.org/10.1109/ICDE.1989.47249. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

SALEM, K.; GARCIA-MOLINA, H. System M: A transaction processing testbed for memory resident data. **IEEE Transactions on Knowledge and Data**, v. 2, n. 1, p. 161–172, 1990. Available in: https://doi.org/10.1109/69.50911. Access in: Sat, 20 May 2017 00:24:24 +0200.

SALLES, M. A. V.; CAO, T.; SOWELL, B.; DEMERS, A. J.; GEHRKE, J.; KOCH, C.; WHITE, W. M. An evaluation of checkpoint recovery for massively multiplayer online games. **Proceedings of the VLDB Endowment**, v. 2, n. 1, p. 1258–1269, 2009. Available in: http://www.vldb.org/pvldb/vol2/vldb09-387.pdf. Access in: Sat, 25 Apr 2020 13:59:39 +0200.

SAUER, C. **Modern techniques for transaction-oriented database recovery**. Tese (Doutorado) – Kaiserslautern University of Technology, Germany, 2017. Available in: http://www.dr.hut-verlag.de/978-3-8439-3297-4.html. Access in: Wed, 12 Feb 2020 16:41:37 +0100.

SAUER, C. Modern techniques for transaction-oriented database recovery. In: GRUST, T.; NAUMANN, F.; BÖHM, A.; LEHNER, W.; HÄRDER, T.; RAHM, E.; HEUER, A.; KLETTKE, M.; MEYER, H. (Ed.). **Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings**. Gesellschaft für Informatik, Bonn, 2019. (LNI, P-289), p. 487–496. Available in: https://doi.org/10.18420/btw2019-30. Access in: Wed, 13 Jan 2021 11:37:30 +0100.

SAUER, C. **GitHub - caetanosauer/zero: Fork of the Shore-MT storage manager used by the research project Instant Recovery**. 2022. Available in: https://github.com/caetanosauer/zero. Access in: February 24, 2022.

SAUER, C.; GRAEFE, G.; HÄRDER, T. Single-pass restore after a media failure. In: SEIDL, T.; RITTER, N.; SCHÖNING, H.; SATTLER, K.; HÄRDER, T.; FRIEDRICH, S.; WINGERATH, W. (Ed.). **Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings**. GI, 2015. (LNI, P-241), p. 217–236. Available in: http://subs.emis.de/LNI/Proceedings/Proceedings241/article11.html. Access in: Thu, 14 Nov 2019 16:35:26 +0100.

SAUER, C.; GRAEFE, G.; HÄRDER, T. Instant restore after a media failure. In: KIRIKOVA, M.; NØRVÅG, K.; PAPADOPOULOS, G. A. (Ed.). **Advances in Databases and Information Systems - 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings**. Springer, 2017. (Lecture Notes in Computer Science, v. 10509), p. 311–325. Available in: https://doi.org/10.1007/978-3-319-66917-5_21. Access in: Tue, 14 May 2019 10:00:53 +0200.

SAUER, C.; GRAEFE, G.; HÄRDER, T. Fineline: log-structured transactional storage and recovery. **Proceedings of the VLDB Endowment**, v. 11, n. 13, p. 2249–2262, 2018. Available in: http://www.vldb.org/pvldb/vol11/p2249-sauer.pdf. Access in: Sat, 25 Apr 2020 13:59:25 +0200.

SBBD2021. **36th Brazilian Symposium on Databases**. 2021. Available in: https://sbbd.org.br/2021/full-papers-ts/. Access in: November 23, 2021.

SCHROEDER, B.; GIBSON, G. A. Understanding failures in petascale computers. In: IOP PUBLISHING. **Journal of Physics: Conference Series**. [*S. l.*], 2007. v. 78, n. 1, p. 012022.

SCHUSTER, A. Introducing the microsoft vista event log file format. **digital investigation**, Elsevier, v. 4, p. 65–72, 2007.

SCHWALB, D.; BERNING, T.; FAUST, M.; DRESELER, M.; PLATTNER, H. nvm malloc: Memory allocation for NVRAM. In: BORDAWEKAR, R.; LAHIRI, T.; GEDIK, B.; LANG, C. A. (Ed.). **International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015**. [*S. n.*], 2015. p. 61–72. Available in:

http://www.adms-conf.org/2015/adms15_schwalb.pdf. Access in: Thu, 12 Mar 2020 11:33:40 +0100.

SCHWALB, D.; FAUST, M.; DRESELER, M.; FLEMMING, P.; PLATTNER, H. Leveraging non-volatile memory for instant restarts of in-memory database systems. In: **32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016**. IEEE Computer Society, 2016. p. 1386–1389. Available in: https://doi.org/10.1109/ICDE.2016.7498351. Access in: Wed, 16 Oct 2019 14:14:56 +0200.

SCHWALB, D.; KUMAR, G.; DRESELER, M.; S., A.; FAUST, M.; HOHL, A.; BERNING, T.; MAKKAR, G.; PLATTNER, H.; DESHMUKH, P. Hyrise-nv: Instant recovery for in-memory databases using non-volatile memory. In: NAVATHE, S. B.; WU, W.; SHEKHAR, S.; DU, X.; WANG, X. S.; XIONG, H. (Ed.). **Database Systems for Advanced Applications - 21st International Conference, DASFAA 2016, Dallas, TX, USA, April 16-19, 2016, Proceedings, Part II**. Springer, 2016. (Lecture Notes in Computer Science, v. 9643), p. 267–282. Available in: https://doi.org/10.1007/978-3-319-32049-6_17. Access in: Fri, 03 Apr 2020 17:07:24 +0200.

SHASHA, D. E. What good are concurrent search structure algorithms for databases anyway? **IEEE Database Eng. Bull.**, v. 8, n. 2, p. 84–90, 1985. Available in: http://sites.computer.org/debull/85JUN-CD.pdf. Access in: Tue, 10 Mar 2020 16:23:50 +0100.

SHASHA, D. E.; GOODMAN, N. Concurrent search structure algorithms. **ACM Transactions on Database Systems (TODS)**, v. 13, n. 1, p. 53–90, 1988. Available in: https://doi.org/10.1145/42201.42204. Access in: Tue, 06 Nov 2018 12:51:47 +0100.

SHORE-MT. **EPFL Official Shore-MT Page**. 2021. Available in: https://sites.google.com/view/shore-mt/. Access in: February 24, 2021.

SIKKA, V.; FÄRBER, F.; LEHNER, W.; CHA, S. K.; PEH, T.; BORNHÖVD, C. Efficient transaction processing in SAP HANA database: the end of a column store myth. In: CANDAN, K. S.; CHEN, Y.; SNODGRASS, R. T.; GRAVANO, L.; FUXMAN, A. (Ed.). **Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012**. ACM, 2012. p. 731–742. Available in: https://doi.org/10.1145/2213836.2213946. Access in: Tue, 06 Nov 2018 11:07:37 +0100.

STONEBRAKER, M. Managing persistent objects in a multi-level store. In: CLIFFORD, J.; KING, R. (Ed.). **Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, USA, May 29-31, 1991**. ACM Press, 1991. p. 2–11. Available in: https://doi.org/10.1145/115790.115791. Access in: Tue, 06 Nov 2018 11:07:39 +0100.

STONEBRAKER, M.; WEISBERG, A. The voltdb main memory DBMS. **IEEE Database Engineering Bulletin**, v. 36, n. 2, p. 21–27, 2013. Available in: http://sites.computer.org/debull/A13june/VoltDB1.pdf. Access in: Tue, 10 Mar 2020 16:23:49 +0100.

STRICKLAND, J. P.; UHROWCZIK, P. P.; WATTS, V. L. IMS/VS: an evolving system. **IBM Systems Journal**, v. 21, n. 3, p. 490–510, 1982. Available in: https://doi.org/10.1147/sj.214.0490. Access in: Fri, 13 Mar 2020 14:38:34 +0100.

STRUKOV, D. B.; SNIDER, G. S.; STEWART, D. R.; WILLIAMS, R. S. The missing memristor found. **Nature**, Nature Publishing Group, v. 453, n. 7191, p. 80–83, 2008. Available in: https://doi.org/10.1038/nature06932. Access in: July 22, 2020.

TAFT, R.; MANSOUR, E.; SERAFINI, M.; DUGGAN, J.; ELMORE, A. J.; ABOULNAGA, A.; PAVLO, A.; STONEBRAKER, M. E-store: Fine-grained elastic partitioning for distributed transaction processing. **Proceedings of the VLDB Endowment**, v. 8, n. 3, p. 245–256, 2014. Available in: http://www.vldb.org/pvldb/vol8/p245-taft.pdf. Access in: Sat, 25 Apr 2020 13:58:53 +0200.

TAN, K.; CAI, Q.; OOI, B. C.; WONG, W.; YAO, C.; ZHANG, H. In-memory databases: Challenges and opportunities from software and hardware perspectives. **ACM Sigmod Record**, v. 44, n. 2, p. 35–40, 2015. Available in: https://doi.org/10.1145/2814710.2814717. Access in: Fri, 06 Mar 2020 21:56:30 +0100.

Tang Yanjun; Luo Wen-hua. A model of crash recovery in main memory database. In: **2010 International Conference On Computer Design and Applications**. [*S. l.*: *s. n.*], 2010. v. 5, p. V5–206–V5–207.

TAY, Y. C.; GOODMAN, N.; SURI, R. Locking performance in centralized databases. **ACM Transactions on Database Systems (TODS)**, v. 10, n. 4, p. 415–462, 1985. Available in: https://doi.org/10.1145/4879.4880. Access in: Tue, 06 Nov 2018 12:51:47 +0100.

TEAM, T. In-memory data management for consumer transactions the times-ten approach. In: DELIS, A.; FALOUTSOS, C.; GHANDEHARIZADEH, S. (Ed.). **SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA**. ACM Press, 1999. p. 528–529. Available in: https://doi.org/10.1145/304182.304244. Access in: Tue, 06 Nov 2018 11:07:38 +0100.

THOMSON, A.; DIAMOND, T.; WENG, S.; REN, K.; SHAO, P.; ABADI, D. J. Calvin: fast distributed transactions for partitioned database systems. In: CANDAN, K. S.; CHEN, Y.; SNODGRASS, R. T.; GRAVANO, L.; FUXMAN, A. (Ed.). **Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012**. ACM, 2012. p. 1–12. Available in: https://doi.org/10.1145/2213836.2213838. Access in: Tue, 06 Nov 2018 11:07:39 +0100.

TIMESTEN. **Oracle TimesTen In-Memory Database Documentation**. 2020. Accessed in December 1, 2020. Available in: https://docs.oracle.com/database/timesten-18.1/. Access in: December 1, 2020.

TU, S.; ZHENG, W.; KOHLER, E.; LISKOV, B.; MADDEN, S. Speedy transactions in multicore in-memory databases. In: KAMINSKY, M.; DAHLIN, M. (Ed.). **ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013**. ACM, 2013. p. 18–32. Available in: https://doi.org/10.1145/2517349.2522713. Access in: Tue, 06 Nov 2018 16:59:32 +0100.

TUCKER, A. B. **Computer science handbook**. [*S. l.*]: CRC press, 2004.

VOLDB DOCUMENTATION. **VolDB Documentation**. 2020. Available in: https://docs.voltdb.com. Access in: December 12, 2020.

WANG, T.; JOHNSON, R. Scalable logging through emerging non-volatile memory. **Proceedings of the VLDB Endowment**, v. 7, n. 10, p. 865–876, 2014. Available in: http://www.vldb.org/pvldb/vol7/p865-wang.pdf. Access in: Sat, 25 Apr 2020 13:59:34 +0200.

WANG, Z.; QIAN, H.; LI, J.; CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In: BULTERMAN, D. C. A.; BOS, H.; ROWSTRON, A. I. T.; DRUSCHEL, P. (Ed.). **Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014**. ACM, 2014. p. 26:1–26:15. Available in: https://doi.org/10.1145/2592798.2592815. Access in: Mon, 31 Aug 2020 18:56:20 +0200.

WEIKUM, G.; VOSSEN, G. **Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery**. [*S. l.*]: Morgan Kaufmann, 2002. ISBN 1-55860-508-8. Access in: Mon, 28 Aug 2006 07:47:06 +0200.

WIESMANN, M.; SCHIPER, A.; PEDONE, F.; KEMME, B.; ALONSO, G. Database replication techniques: A three parameter classification. In: **19th IEEE Symposium on Reliable Distributed Systems, SRDS'00, Nürnberg, Germany, October 16-18, 2000, Proceedings**. IEEE Computer Society, 2000. p. 206–215. Available in: https://doi.org/10.1109/RELDI.2000.885408. Access in: Wed, 16 Oct 2019 14:14:49 +0200.

WILLHALM, T.; OUKID, I.; MÜLLER, I.; FAERBER, F. Vectorizing database column scans with complex predicates. In: BORDAWEKAR, R.; LANG, C. A.; GEDIK, B. (Ed.). **International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2013, Riva del Garda, Trento, Italy, August 26, 2013**. [*S. n.*], 2013. p. 1–12. Available in: http://www.adms-conf.org/2013/muller_adms13.pdf. Access in: Thu, 12 Mar 2020 11:33:40 +0100.

WILLHALM, T.; POPOVICI, N.; BOSHMAF, Y.; PLATTNER, H.; ZEIER, A.; SCHAFFNER, J. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. **Proceedings of the VLDB Endowment**, v. 2, n. 1, p. 385–394, 2009. Available in: http://www.vldb.org/pvldb/vol2/vldb09-327.pdf. Access in: Sat, 25 Apr 2020 13:59:04 +0200.

WINTER, P. R. P6 file corruption. 2018.

WU, Y.; GUO, W.; CHAN, C.; TAN, K. Fast failure recovery for main-memory dbmss on multicores. In: SALIHOGLU, S.; ZHOU, W.; CHIRKOVA, R.; YANG, J.; SUCIU, D. (Ed.). **Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017**. ACM, 2017. p. 267–281. Available in: https://doi.org/10.1145/3035918.3064011. Access in: Tue, 06 Nov 2018 11:07:37 +0100.

YADAVA, H. **The Berkeley DB Book**. [*S. l.*]: Apress, 2007.

YAO, C.; AGRAWAL, D.; CHEN, G.; OOI, B. C.; WU, S. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In: ÖZCAN, F.; KOUTRIKA, G.; MADDEN, S. (Ed.). **Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016**. ACM, 2016. p. 1119–1134. Available in: https://doi.org/10.1145/2882903.2915208. Access in: Wed, 14 Nov 2018 10:56:20 +0100.

YU, X.; BEZERRA, G.; PAVLO, A.; DEVADAS, S.; STONEBRAKER, M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. **Proceedings of the VLDB Endowment**, v. 8, n. 3, p. 209–220, 2014. Available in: http://www.vldb.org/pvldb/vol8/p209-yu.pdf. Access in: Sat, 25 Apr 2020 13:59:28 +0200.

ZAMANIAN, E.; YU, X.; STONEBRAKER, M.; KRASKA, T. Rethinking database high availability with RDMA networks. **Proceedings of the VLDB Endowment**, v. 12, n. 11, p.

1637–1650, 2019. Available in: http://www.vldb.org/pvldb/vol12/p1637-zamanian.pdf. Access in: Sat, 25 Apr 2020 13:59:01 +0200.

ZHANG, H.; CHEN, G.; OOI, B. C.; TAN, K.; ZHANG, M. In-memory big data management and processing: A survey. **IEEE Transactions on Knowledge and Data**, v. 27, n. 7, p. 1920–1948, 2015. Available in: https://doi.org/10.1109/TKDE.2015.2427795. Access in: Sat, 20 May 2017 00:24:24 +0200.

ZHANG, J.; YAO, Z.; FENG, J. Ncredis: An nvm-optimized redis with memory caching. In: STRAUSS, C.; KOTSIS, G.; TJOA, A. M.; KHALIL, I. (Ed.). **Database and Expert Systems Applications - 32nd International Conference, DEXA 2021, Virtual Event, September 27-30, 2021, Proceedings, Part II**. Springer, 2021. (Lecture Notes in Computer Science, v. 12924), p. 70–76. Available in: https://doi.org/10.1007/978-3-030-86475-0_7. Access in: Mon, 03 Jan 2022 22:20:12 +0100.

ZHANG, Y.; SWANSON, S. A study of application performance with non-volatile main memory. In: **IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015**. IEEE Computer Society, 2015. p. 1–10. Available in: https://doi.org/10.1109/MSST.2015.7208275. Access in: Wed, 16 Oct 2019 14:14:51 +0200.

ZHANG, Y.; YANG, J.; MEMARIPOUR, A.; SWANSON, S. Mojim: A reliable and highly-available non-volatile memory system. In: ÖZTURK, Ö.; EBCIOGLU, K.; DWARKADAS, S. (Ed.). **Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015**. ACM, 2015. p. 3–18. Available in: https://doi.org/10.1145/2694344.2694370. Access in: Tue, 06 Nov 2018 11:07:42 +0100.

ZHENG, W.; TU, S.; KOHLER, E.; LISKOV, B. Fast databases with fast durability and recovery through multicore parallelism. In: FLINN, J.; LEVY, H. (Ed.). **11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014**. USENIX Association, 2014. p. 465–477. Available in: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_wenting. Access in: Tue, 02 Feb 2021 08:05:58 +0100.