Main Memory Database Recovery: A Survey

ARLINO MAGALHAES, Federal University of Ceara/Federal University of Piaui, Brazil
JOSE MARIA MONTEIRO and ANGELO BRAYNER, Federal University of Ceara, Brazil

Many of today's applications need massive real-time data processing. In-memory database systems have become a good alternative for these requirements. These systems maintain the primary copy of the database in the main memory to achieve high throughput rates and low latency. However, a database in RAM is more vulnerable to failures than in traditional disk-oriented databases because of the memory volatility. DBMSs implement recovery activities (logging, checkpoint, and restart) for recovery proposes. Although the recovery component looks similar in disk- and memory-oriented systems, these systems differ dramatically in the way they implement their architectural components, such as data storage, indexing, concurrency control, query processing, durability, and recovery. This survey aims to provide a thorough review of in-memory database recovery techniques. To achieve this goal, we reviewed the main concepts of database recovery and architectural choices to implement an in-memory database system. Only then, we present the techniques to recover in-memory databases and discuss the recovery strategies of a representative sample of modern in-memory databases.

CCS Concepts: • **Information systems** → *Main memory engines*;

Additional Key Words and Phrases: Main memory, in-memory database, failure recovery

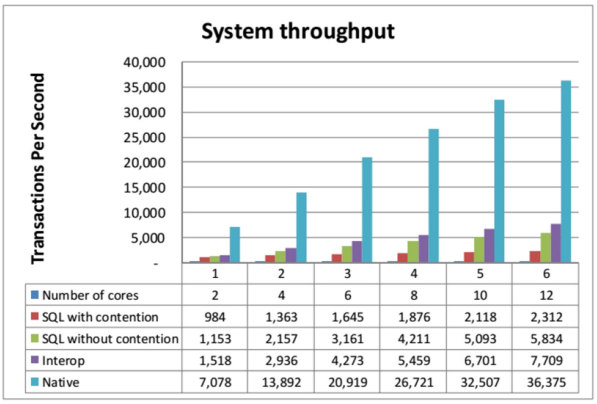
ACM Reference format:
Arlino Magalhaes, Jose Maria Monteiro, and Angelo Brayner. 2021. Main Memory Database Recovery: A Survey. *ACM Comput. Surv.* 54, 2, Article 46 (March 2021), 36 pages.
https://doi.org/10.1145/3442197

1 INTRODUCTION

Main Memory Database - MMDB (or In-Memory Database - IMDB) technology has proved to be an efficient alternative for a real-time view of operational data (OLTP-style) and high-performance mission-critical 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 5 years [185, 206]. Moreover, other recent hardware/architecture improvements can potentially provide better performance to MMDBs with low overhead, such as NUMA architecture [106], SIMD instructions [200], RDMA networking [129], hardware transactional memory [109], and non-volatile memory [8]. Such advances have boosted the development of MMDBs.

Authors' addresses: A. Magalhaes, Federal University of Ceara/Federal University of Piaui, Piaui, Brazil; email: arlino@ufpi.edu.br; J. M. Monteiro and A. Brayner, Federal University of Ceara, Ceara, Brazil; emails: {monteiro, brayner}@dc.ufc.br.

0360-0300/2021/03-ART46 $15.00
https://doi.org/10.1145/3442197



	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

Fig. 1. Hekaton vs. SQL Server scalability experiment [42].

The technological foundations of MMDB arose in the 1980s [41, 45–47, 76, 104]. In this period, some products were developed, such as IMS/Fast Path [60, 182], MARS MMDB [47], System M [167], TPK [113], OBE [19], and HALO [58], just to name a few. Much of this research has focused on improving the performance of traditional disk-based database systems or fitting the entire database into the main memory. However, the high price and limited capacity of RAM at that period made the application of MMDB an infeasible solution. Nonetheless, in the Nineties, the advances in hardware technology re-generated interests in MMDB research [58, 112, 185, 206]. 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 [13, 88], ClustRa [85], TimesTen [96, 189], and P*Time [25].

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 [73, 78, 80]. Some MMDBs give a direct pointer to the main memory to access an object. This is called swizzling [58, 180]. 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 [73, 80].

To illustrate the potential of MMDBs, Diaconu et al. [42] provide 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 reside 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).

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 period 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 [74, 133].

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. This article aims to elucidate the main issues regarding in-memory database recovery. In order to achieve this goal, first, a thorough discussion about classical database recovery mechanisms is provided in Section 2, focusing on ARIES-style recovery, a quite popular algorithm for recovering traditional databases. Thereafter, Section 3 sheds light on different strategies implemented by in-memory databases and some technologies that boosted the development of in-memory databases. Section 4 discusses the techniques for implementing recovery in in-memory databases. Section 5 describes the main features of recovery mechanisms delivered by well-known in-memory database systems. Section 6 concludes this article.

2 DATABASE RECOVERY OVERVIEW

This section aims to present the main concepts of DBMS recovery focusing on disk-resident systems. The objective is to provide a sufficient theoretical basis for the next sections. Readers who are already aware of this topic can skip this section. We briefly review 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, modern recovery techniques, 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. To achieve this indivisibility, a database system must maintain the four ACID properties. ACID is an acronym derived from the first letter of each of the four features: atomicity, consistency, isolation, and durability [74]. A brief explanation of ACID properties follows below:

- *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 the 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 [74, 133].

For performance reasons, most DBMSs adopt steal and no-force buffer manager page replacement policies. The buffer manager is the DBMS component that manages the database buffer pool and reads/writes pages from/to the secondary memory. The steal approach is used when the buffer

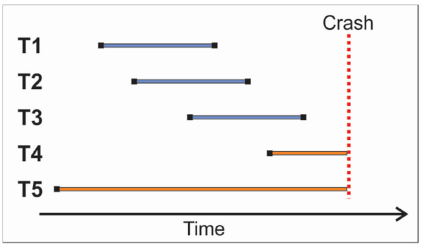


Fig. 2. A database crash scenario [74].

manager protocol allows flushing dirty pages to secondary storage before the transaction commitment. 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 I/O cost to flush pages multiple times [74, 133].

Most recovery techniques do not interfere in buffer manager protocols, i.e., they support steal and no-force approaches. This scenario makes database recovery costs 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 [45, 74, 133]. Figure 2 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 [74].

2.1 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 [74]. 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 cannot 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 [74].

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 [66, 74, 133].

A media crash is a secondary memory failure, such as an 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 [74]. If the specific location of the media failure can be identified, a redo of only the affected area would be warranted. Graefe and Kuno [62] 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.

2.2 Recovery Method

Most database systems implement the Write-Ahead Logging (WAL) [133] 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 [133].

The system writes log records for each of the following update actions [133]:

- *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 [133]:

- *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 [125, 133, 201].

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 [125, 201]. However, for performance reasons, commercial disk-based DBMSs, such as MySQL [140] and Oracle [146], 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 [67, 133, 191].

Before the ARIES algorithm, database recovery seemed to be dominated by the Undo/Redo algorithm, described by Bernstein et al. [16]. After a system failure, the Undo/Redo algorithm starts the recovery scanning backward log, undoing the actions of loser transactions. Then a progressive scan redoes the effects of committed transactions that did not flush to secondary memory. This recovery technique does not interfere in the buffer manager policy, i.e., it supports steal and no-force protocols. Undo/Redo algorithm does not support a fine-granularity locking (e.g., tuple-level). It supports only coarse-granularity locking (e.g., page-level). If a locking level of less than one page is implemented, the Undo phase can cause the Redo phase to lose log tracking of which actions need to be redone [16, 74, 133].

Some DBMSs, such as System R [66] and DB2/MVS V1 [35], first perform the Undo pass and then the Redo pass. This approach is called selective redo paradigm. According to Haerder et al. [74], 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 [74, 133].

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 [74, 133].

In the TCC technique, 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 is recorded in the log. Thus, after a crash, the recovery manager can disregard the log records before the checkpoint. The ACC approach pauses transaction processing; flushes all dirty pages; and, finally, records a checkpoint in the log. When a failure occurs, the recovery manager can only redo the transaction's actions after the checkpoint. TCC and ACC cause system performance degradation because they pause transaction performing during the execution of the checkpoint [74, 119, 133].

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

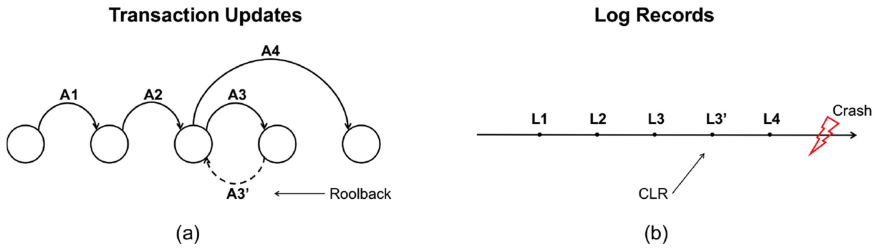


Fig. 3. CLR generated by a partial rollback [133].

checkpoint costs. Instead, the fuzzy approach stores information about the buffer occupation (the active transactions, their states, and the addresses of their most recently written log records, and also the dirty pages) rather than pages to the secondary memory. When the fuzzy process finishes, a checkpoint record is written on the log. After a crash, the system reads checkpoint information and log records forward from the last checkpoint to know the buffer occupation at the crash moment. The fuzzy checkpoint 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 [74, 119, 133].

2.3 ARIES Algorithm

ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) [133] was designed to support steal and no-force buffer approaches, fine-granularity, WAL, 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.

2.3.1 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 [132, 133]. 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 [32], DB2/MVS V1 [30], and NonStop SQL [69], undo the same CLR one or more times in face of multiple failures.

Figure 3 shows an example of a CLR generated by a partial rollback. Figure 3(a) illustrates updates of a transaction T1 and Figure 3(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 [132, 133].

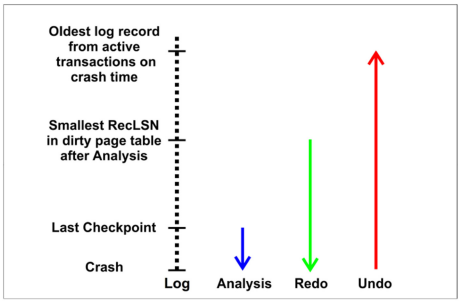


Fig. 4. ARIES phases over the log during the recovery process [155].

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 [134] and ARIES/KVL [130], are not the exact inverse of their undo operations. This is very efficient during recovery. Furthermore, 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 [132, 133].

ARIES implemented the concept of Nested Top Actions (NTAs). An NTA starts inside another action (parent). However, an NTA has no special relationship with its parent. For example, an NTA cannot 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 [120]. 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 [132, 133, 164].

2.3.2 Restart Recovery. ARIES recovery schema consists of three phases (analysis, redo, and undo). We summarize the ARIES phases below and Figure 4 illustrates the execution of these phases over the log [132, 133].

(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.

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 dirty table represents information about dirty buffer pages. Then, 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 oldest log record of the dirty pages [132–134, 164].

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 (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., 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 [132–134, 164].

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 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 [132–134, 164].

2.4 Variant ARIES Algorithms

This section exposes the main aspects of some algorithms that extend the original ARIES algorithm: ARIES/NT, ARIES-RRH, ARIES/IM, ARIES/LHS, and ARIES/CSA.

2.4.1 ARIES/NT. The original ARIES algorithm was designed to work with single-level transactions. ARIES/NT (ARIES for Nested Transactions) [164] 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 [123]. 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 [132, 164].

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 [132, 164].

2.4.2 ARIES-RRH. ARIES-RRH (ARIES with Restricted Repeating of History) [136] 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 2.2) performs the Undo phase before the Redo phase, but it does not support fine-granularity locking [132, 136].

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 CPU processing during the recovery, but it still maintains all the good features of the original ARIES algorithm [132, 136].

2.4.3 ARIES/IM. ARIES/IM (ARIES for Index Management) [134] 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 [66] 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 [132, 134, 161].

It is worth mentioning that a previous work called *ARIES/KVL* [130] addressed concurrency control and recovery of B-tree indexes. However, the concurrency enhancements of ARIES/KVL were still considered inadequate. Mohan [130] discussed only the possibility of correct logging and recovery in the face of concurrent executions. The article did not implement logging and recovery in ARIES/KVL. Works before ARIES/IM and KVL researched recovery with concurrency control (e.g., [14], [128], [165], [177], and [178]) but they did not consider concurrency control and B-trees indexes with fine-granularity locking in face of failures.

2.4.4 ARIES/LHS. Linear Hashing with Separators (LHS) [99] 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/LHS (ARIES for Linear Hashing with Separators) [131] 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 [131, 132].

2.4.5 ARIES/CSA. ARIES/CSA (ARIES for the Client-Server Architecture) [135] 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 cannot 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 [132, 135].

2.5 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 [62, 169]. For example, flash storage promises higher performance than magnetic disks. However, it also presents its problems, such as limited endurance and write/read asymmetry [17, 23]. The failure of individual pages in storage is described in [62] 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.

Single-page repair [62] handles the recovery of individual pages (after single-page failure) instead of recovering the whole media device. This technique is executed during system running 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 [170] allows the backup restoration and log replay in a single operation after a media failure. During the 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 number of partitions can be reduced by intermediate merges to speed up the recovery process.

Instant restart [61, 77] is a technique that enables new transactions to be performed 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 [168, 171] presents a technique to restore a database incrementally and on-demand after a media failure, immediately after the database is restarted. This technique allows transaction processing during the recovery process. It sorts and indexes the records in a partitioned indexed log during transaction processing. The log partitions have the same idea as the instant restart technique. After a media failure, the 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 also recovers pages on-demand for transactions. A new transaction can perform as soon as its necessary pages are restored. The log partitions are periodically merged during normal system execution to accelerate the page recovery.

2.6 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 [198, 205]. According to Gray et al. [65], 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 [65, 198, 205].

Paxos [97, 98], ZAB (ZooKeeper Atomic Broadcast) [89], and Raft [144] 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 [64] 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. For this reason, replication is not covered in this article [171].

3 MAIN 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. Readers already aware of this topic can skip this section. 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 [49, 206].

3.1 Data Storage

MMDBs avoid the page-based indirection through a buffer pool. Disk-resident databases use a buffer pool to access records. 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. MMDBs are not dependent on disk logical layout (e.g., page size). Thereby, they can store data in any format in order to achieve high throughput 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 [49, 80, 100, 154].

Physical pointers, instead of buffer pool indirection, can improve access to records in memory in order of magnitude for MMDBs. Lehman et al. [105] highlighted 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 [187]). 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 [27], and Oracle TimesTen) can access any record in the database. This approach prevents the balance of partitions accessed frequently [49, 184].

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. Section 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. MMDBs can generate snapshots by an asynchronized consistent checkpoint, i.e., the system does not need to pause transaction processing to create a snapshot [22, 125, 142]. Section 4.3 discusses checkpoint techniques for MMDBs.

An organizational choice is whether the database is row- or column-oriented [1]. 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. 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 analytical 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 [10, 34].

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 [10, 34]. SAP HANA explores a tuple propagation schema from row to columnar storage to support HTAP [179]. HyPer uses a virtual memory schema to perform OLTP and OLAP transactions on the same database [56].

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., [12], [15], [16], [67], [68], [82], [137], [197], and [204]). Several researches compared the tradeoffs between pessimistic and optimistic concurrency control protocols, such as [2], [3], [24], [54], and [186]. Pessimistic concurrency control (PCC) assumes that concurrency conflicts will occur frequently and tries to avoid them during the performing, blocking, and aborting transactions [197]. 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 [95].

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 [16, 197]. 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 [80, 206]. Garcia-Molina and Salem [58] propose a "lock bit" added to each object to indicate whether the object is blocked or not. In [160], Ren et al. 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. Usually, the checkpoint component acts as a garbage collector. 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 [49, 121, 142]. Agrawal et al. [2] 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 [102, 206].

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 [90, 181, 187].

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 [72]. 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 3.1). Therefore, indexes can store direct pointers to records, instead of primary keys [5].

Rosenblum et al. [162] 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) [18], Cache Sensitive Search Trees (CSS$^+$-Trees) [157], Cache Sensitive B$^+$-Trees (CSB$^+$-Trees) [157], Prefetching B$^+$-Trees (pB$^+$-Trees) [28], Fast Architecture Sensitive Tree (FAST) [93], and Adaptive Radix Tree (ART) [108] are indexing techniques for cache efficiency. Hash indexes also can be further optimized for better cache utilization [50].

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 [185]. 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 [148]. Bw-tree [111] and Mass-Tree [126] 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) [107, 163], T-Tree [103], Δ-Tree [36], and BD-Tree [37].

3.4 Query Processing

MMDBs avoid the traditional iterator model (Volcano-style processing) [63]. 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 [49, 80, 206]. 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 [42, 92, 127]. For example, a transaction must be a single stored procedure in VoltDB [125, 181].

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 4 explains in more detail the main memory database system recovery.

3.6 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 unviable. 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 [185, 206]. This section presents the core technologies that leverage MMDBs.

3.6.1 Non-Uniform Memory Access (NUMA). The CPU speed grew faster than the main memory speed in the last decades [143]. 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. 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 [106, 122]. In the context of DBMS research, NUMA-awareness cares about minimizing the accesses to remote NUMA domains by data partitions (e.g., [6], [20], [40], [106], and [122]); managing system memory accesses on latency-sensitive workloads (e.g., OLTP workloads) since NUMA has heterogeneous access latency (e.g., [152] and [153]); and transferring the data across NUMA domains efficiently, i.e., data shuffling (e.g., [117]).

3.6.2 Hardware Transactional Memory (HTM). 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 [81, 91, 109, 124]. HyPeR [109] and DBX [196] databases explore HTM to achieve concurrency. Karnagel et al. [91], Makreshanski et al. [124], and Wang et al. [196] improve in-memory database index performance with HTM.

3.6.3 Non-Volatile Memory (NVM). Advanced NVM technologies include Phase-Change Memory (PCM) [158], Memristors [183], and Spin-Transfer Torque Magnetic RAM (STTMRAM) [44]. 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 [8, 9, 38]. Although there are many researches that describe the DBMS architecture affected by NVM (e.g., [8], [9], [11], [29], [59], [79], [147], [174], [192], [193], [207]), 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. Although the focus of this survey is not on databases that adopt NVM, the topic is interesting so that readers can learn and compare different database recovery strategies [8, 9, 38]. Section 4.5 very briefly discusses the recovery in NVM-resident database systems.

3.6.4 Single Instruction Multiple Data (SIMD). 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 [141, 200]. An efficient MMDB design should consider data-level parallelism. SIMD instructions can speed up expensive database operations (e.g., join and sort) [31, 141]. SAP HANA implements a vector schema by SIMD to speed up dictionary decompression during scans [199]. SIMD can improve vector-style computation commonly applied in Big Data analytics [139, 156].

3.6.5 Remote Direct Memory Access (RDMA). 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 [129]. FaRM implements lock-free reads over RDMA [43]. Hyper can generate backup files in stable storage via RDMA to relieve the server CPU of the data transmission task [92].

4 MMDB 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, in which we discussed key database recovery concepts. Throughout this article, 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 systems. However, these systems are very different in several details. For example, ARIES-style recovery protocols are avoided in MMDBs for performance reasons [49, 87].

4.1 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 [45, 202].

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 no 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. 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 [45, 116, 202].

4.2 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 [58]. 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 [203].

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. Most MMDBs prefer SSD as the log device to increase I/O performance. Besides, several systems also avoid logging indexes. After a crash, indexes can be rebuilt in parallel to recovery [125, 201, 208]. For this reason, this paper 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 [125, 201].

MMDBs can use the group commit and pre-commit techniques to improve secondary memory access for writing log records. In group commit [41, 75], 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 [41, 58], 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.

4.3 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) [42, 45, 58, 181].

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 [118, 159, 166].

Many MMDBs perform the Copy-on-Update (COU) snapshot algorithm [41, 45]. 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 [22, 125]. COU has many variants, such as in [53], [118], and [194].

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 [22, 125]. There are also other proposed snapshot algorithms such as Naive [21, 173], Zigzag [22], PingPong [22], Hourglass [114, 115], and Piggyback [114, 115].

4.4 Restart

After a system crash, the MMDB must reload the last snapshot from secondary storage into the main memory and, then, redo log records from the checkpoint record. During recovery, as the system is offline (i.e., the database can not process new transactions), system performance degrades. Efficient reload and log replay approaches are essential to meet the expectations of high performance [70, 71].

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 [70, 71].

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 [42]. SilioR implements a highly parallel log replay schema that prevents unnecessary allocations, copies, and work [208]. 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 [201]. RAMCloud [145] uses a log-structured

approach (similar to log-structured file systems [163]) 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 [110]. For instance, the Anti-caching [39] from H-Store and Siberia [7, 48, 110] 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 lower 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 [49, 125, 201].

4.5 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 [8, 11].

Write-behind Logging (WBL) [11] 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 [8, 11].

A Dirty Tuple Table (DTT) is used to track transaction updates. Each DTT entry has the transaction ID, the table modified, and a metadata 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 [8, 11].

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 [8, 11].

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

Table 1. Some Modern MMDBs and Their Main Recovery Features

	Concurrence Control	Logging	Checkpoint	Instant recovery
Hekaton [42, 55, 101]	optimistic MVCC	operation logging	delta and data files	no
VoltDB [125, 181]	serial execution in partitions	command logging	snapshot	no
HyPer [56, 92, 138]	serial execution in partitions for OLTP, virtual snapshot for OLAP	command logging	snapshot	no
SAP HANA [51, 52, 179]	MVCC, 2PC	operation logging	snapshot	no
SiloR [208]	optimistic MVCC	value logging	"fuzzy" snapshot	no
TimesTen [96, 188, 189]	MVCC, 2PL	value logging	"fuzzy" checkpoint, snapshot	no
PACMAN [201]	serial execution in partitions	command logging	-	no
Adaptive Logging [203]	serial execution in partitions	command logging, ARIES logging	snapshot	no
FineLine [168, 169, 172]	-	physiological logging	-	yes

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 [8, 11].

In the context of database recovery, we can also mention some researches such as NVRAM Group Commit [150], Editable Atomic Writes (EAW) [33], NV-Logging [84], Fast Optimistic Engine for Data Unification Services (FOEDUS) [94], REWIND [26], JUSTDO logging [86], and Hyrise-NV [175]. Other works related to database recovery in NVM are: [4], [9], [57], [84], [150], [176], and [195].

5 MMDB RECOVERY STRATEGIES

MMDBs are implemented in a variety of ways, as discussed in Section 3. 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. This article focuses on relational MMDB recovery. However, 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 3.2), logging (cf. Section 4.2), checkpoint (cf. Section 4.3), and instant recovery (cf. Section 2.5). For each system, we give an overview of the general features of the MMDB and, then discuss in-depth the methods of logging, checkpoint, and recovery. In the recovery approaches covered, only FineLine implements an instant recovery approach. The only way for these other systems to be available during recovery is to adopt some high-availability strategy.

5.1 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 T-SQL (Transact-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 [42, 55, 101].

The log in Hekaton is logical and redo-only. A transaction generates log records only at the commit time since Hekaton does not implement the 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 [42].

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 [42].

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 highly parallelizable recovery process. 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 [42].

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 [42].

5.2 VoltDB

VoltDB is an OLTP MMDB designed from H-Store [90] (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 [125, 181].

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 [125, 181].

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

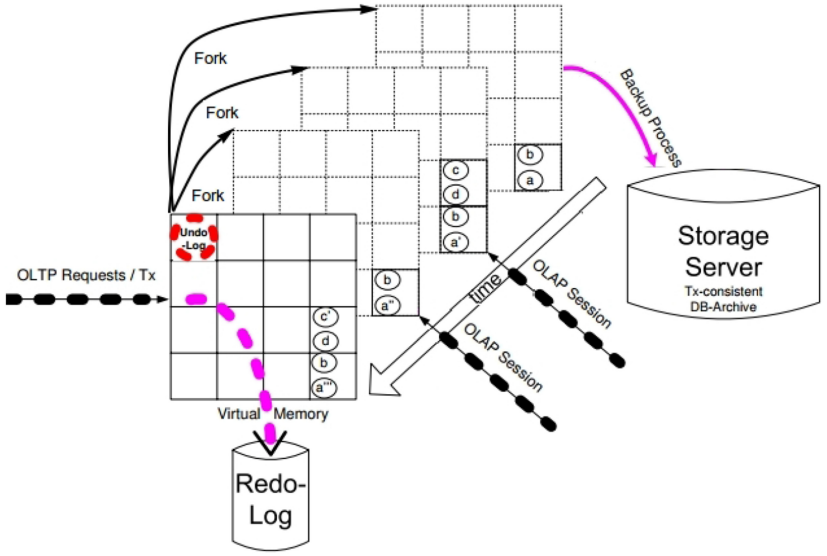


Fig. 5. HyPer's virtual memory schema to support OLTP and OLAP transactions performing in the same database [56].

appropriate sites. This recovery schema can work even if the site topology changes. For example, when a site cannot 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 [125, 181].

5.3 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 5. HyPer supports multiple OLAP sessions (one session per core) [49, 56].

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 5 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) [56, 92, 138].

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

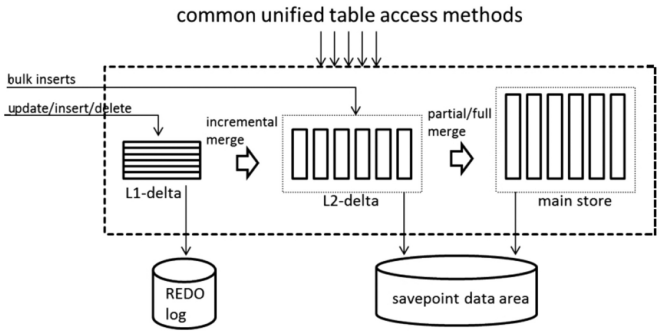


Fig. 6. Unified table structure schema in SAP HANA [179].

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 [56, 92, 138].

HyPer employs redo-only command logging (on the bottom of Figure 5) 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 5). 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 [56, 92, 138].

5.4 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 6) 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 [49, 51, 52, 179].

SAP HANA was implemented for OLAP workloads originally. This kind of workload is very appropriate 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 store 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 [52, 179].

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 6 outlines some of the details of SAP HANA persistence. After a crash, the system loads the savepoint and perform REDO log actions [52, 179].

5.5 SiloR

SiloR [208] added a concurrent recovery schema in Silo [190]. 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), loggers, and checkpointers [208].

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 stored on the 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 replay 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 that were 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 [208].

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, $n \times m$ 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 [208].

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 [208].

5.6 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 [83, 96, 188, 189].

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 [96, 188, 189].

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 [96, 188, 189].

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 [96, 188, 189].

5.7 PACMAN

PACMAN [201] 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 [149, 151], 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 [201].

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 [201].

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

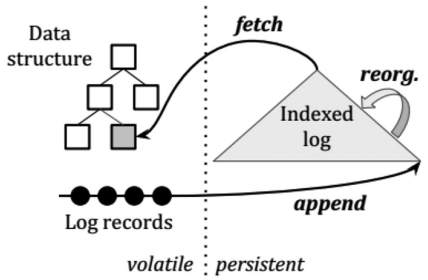


Fig. 7. FineLine propagation schema [172].

values. Second, it allows inter-batch parallel executions in which different log batches are replayed in parallel by applying a pipelined execution optimization [201].

5.8 Adaptive Logging

Adaptive Logging [203] 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 tradeoff 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 a dependency relationship can process concurrently and the others 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 [203].

5.9 FineLine

FineLine [168, 169, 172] is an in-memory system that proposes a log-structured data structure to provide persistence. The log is a partitioned index by page IDs, like in Instant Restore (see Section 2.5 for more details). FineLine in-memory data structures are key-value stores. However, the system architecture is generalized to support any storage and index structures with any concurrency control protocol. For example, it can be handled as a generic software library or relational storage component. Although the system does not specify any concurrency control schema, the authors of [172] 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 a physical log record. The log is redo-only. Transaction update records are sorted and stored in log partitions at commit time. Periodically, partitions are merged to deliver acceptable read performance. Undo log records are not flushed to secondary memory, 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. The log is the only way to recover the system; checkpoints are not implemented. Figure 7 illustrates the FineLine propagation schema. After a system failure, the system performs an instant recovery schema, i.e., transactions can be

performed during the recovery process. After a system failure, the recovery manager recovers the pages incrementally by traversing the indexed log. 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 [168, 169, 172].

6 CONCLUSION

Although in-memory database systems have been studied since the 80s, only the recent technological improvements have boosted the interest in storing the entirety of the primary database in the main memory. Shifting data storage layers from secondary storage to main memory may result in a rethinking of the design of traditional systems. Thereby, in-memory database systems can allow orders of magnitude performance improvements over disk-resident database systems. Overhead sources that do not exist in disk-resident databases can significantly impair the overall performance of in-memory databases. This scenario transforms the recovery manager into a crucial database component [49, 206].

In this survey, we have focused on durability and recovery techniques for in-memory database systems. Besides describing the logging, checkpoint, and restart algorithms, we also discussed recovery procedures implemented by a representative subset of modern in-memory database systems. However, the design approaches for in-memory databases are diverse and, consequently, the recovery component can be implemented in diverse ways, according to the architectural choices of the system. For a better understanding for the reader, we highlighted the architectural choices and implementation techniques for in-memory database. Besides, we reviewed the traditional database recovery algorithms focusing on ARIES-style recovery, the most popular technique to recover disk-based database systems.

REFERENCES

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. 967–980.

[2] Rakesh Agrawal, Michael J. Carey, and Miron Livny. 1987. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems (TODS)* 12, 4 (1987), 609–654.

[3] Rakesh Agrawal and David J. Dewitt. 1985. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Transactions on Database Systems (TODS)* 10, 4 (1985), 529–564.

[4] Rakesh Agrawal and H. V. Jagadish. 1989. Recovery algorithms for database machines with non-volatile main memory. In *Proceedings of the International Workshop on Database Machines*. Springer, 269–285.

[5] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a modern processor: Where does time go? In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99) (September 7-10, 1999, Edinburgh, Scotland, UK)*. 266–277.

[6] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively parallel sort-merge joins in main memory multi-core database systems. *Arxiv Preprint Arxiv:1207.0145* (2012).

[7] Karolina Alexiou, Donald Kossmann, and Per-Ake Larson. 2013. Adaptive range filters for cold data: Avoiding trips to Siberia. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1714–1725.

[8] Joy Arulraj and Andrew Pavlo. 2017. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1753–1758.

[9] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 707–722.

[10] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 583–598.

[11] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind logging. *Proceedings of the VLDB Endowment* 10, 4 (2016), 337–348.

[12] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 185–196.

[13] Jerry Baulier, Philip Bohannon, S. Gogate, C. Gupta, and S. Haldar. 1999. DataBlitz storage manager: Main-memory database performance for critical applications. In *ACM SIGMOD Record*, Vol. 28. Association for Computing Machinery (ACM), 519–520.
[14] Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of operations on b-trees. *Acta Informatica* 9, 1 (1977), 1–21.
[15] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.
[16] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Vol. 370. Addison-Wesley New York.
[17] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. 2003. Introduction to flash memory. *Proc. IEEE* 91, 4 (2003), 489–502.
[18] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. Hot: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*. 521–534.
[19] Dina Bitton, Maria Butrico Hanrahan, and Carolyn Turbyfill. 1987. Performance of complex queries in main memory database systems. In *Proceedings of the 1987 IEEE 3rd International Conference onData Engineering*. IEEE, 72–81.
[20] Haran Boral, William Alexander, Larry Clay, George Copeland, Scott Danforth, Michael Franklin, Brian Hart, Marc Smith, and Patrick Valduriez. 1990. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge & Data Engineering* 1 (1990), 4–24.
[21] Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. 2006. Recent advances in checkpoint/recovery systems. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 8–15.
[22] Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. 2011. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. Association for Computing Machinery (ACM), 265–276.
[23] Paulo Cappelletti, Carla Golla, Piero Olivo, and Enrico Zanoni. 2013. *Flash Memories*. Springer Science & Business Media.
[24] Michael J. Carey. 1984. *The Performance of Concurrency Control Algorithms for Database Management Systems*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.
[25] Sang K. Cha and Changbin Song. 2004. P* TIME: Highly scalable OLTP DBMS for managing update-intensive stream workload. In *Proceedings of the 30th International Conference on Very Large Data Bases-Volume 30*. VLDB Endowment, 1033–1044.
[26] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment* 8, 5 (2015), 497–508.
[27] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. 2016. The MEMSQL query optimizer: A modern optimizer for real-time analytics in a distributed database. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1401–1412.
[28] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. 2001. Improving index performance through prefetching. *ACM SIGMOD Record* 30, 2 (2001), 235–246.
[29] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking database algorithms for phase change memory.. In *CIDR*, Vol. 11. 5th.
[30] Josephine M. Cheng, Christopher R. Loosley, Akira Shibamiya, and Patricia S. Worthington. 1984. IBM database 2 performance: Design, implementation, and tuning. *IBM Systems Journal* 23, 2 (1984), 189–210.
[31] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1313–1324.
[32] Brian E. Clark and Michael J. Corrigan. 1989. Application system/400 performance characteristics. *IBM Systems Journal* 28, 3 (1989), 407–423.
[33] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. 2013. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 197–212.
[34] George P. Copeland and Setrag N. Khoshafian. 1985. A decomposition storage model. *ACM SIGMOD Record* 14, 4 (1985), 268–279.
[35] Richard A. Crus. 1984. Data recovery in IBM database 2. *IBM Systems Journal* 23, 2 (1984), 178–188.
[36] Bin Cui, Beng Chin Ooi, Jianwen Su, and Kian-Lee Tan. 2003. Contorting high dimensional data for efficient main memory KNN processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. 479–490.
[37] Bin Cui, Beng Chin Ooi, Jianwen Su, and K.-L. Tan. 2004. Main memory indexing: The case for BD-tree. *IEEE Transactions on Knowledge and Data Engineering* 16, 7 (2004), 870–874.

[38] Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael Stonebraker, Stan Zdonik, and Subramanya Dulloor. 2014. A prolegomenon on OLTP database systems for non-volatile memory. *ADMS@ VLDB* (2014).
[39] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1942–1953.
[40] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. 1990. The gamma database machine project. *IEEE Trans. Knowl. Data Eng.* 2, 1 (1990), 44–62.
[41] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. 1984. *Implementation Techniques for Main Memory Database Systems*. Vol. 14. Association for Computing Machinery.
[42] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery (ACM), 1243–1254.
[43] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th {USENIX} Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.
[44] Alexander Driskill-Smith. 2010. Latest advances and future prospects of STT-RAM. In *Proceedings of the Non-Volatile Memories Workshop*. 11–13.
[45] Margaret H. Eich. 1986. Main memory database recovery. In *Proceedings of the Fall Joint Computer Conference*. IEEE Computer Society Press, 1226–1232.
[46] Margaret H. Eich. 1987. A classification and comparison of main memory database recovery techniques. In *Proceedings of the 1987 IEEE 3rd International Conference on Data Engineering*. IEEE, 332–339.
[47] Margaret H. Eich. 1988. Mars: The design of a main memory database machine. *Database Machines and Knowledge Base Machines* 43 (1988), 325–338.
[48] Ahmed Eldawy, Justin Levandoski, and Per-Ake Larson. 2014. Trekking through Siberia: Managing cold data in a memory-optimized database. *Proceedings of the VLDB Endowment* 7, 11 (2014), 931–942.
[49] Franz Faerber, Alfons Kemper, Per-Ake Larson, Justin Levandoski, Thomas Neumann, Andrew Pavlo, et al. 2017. Main memory database systems. *Foundations and Trends® in Databases* 8, 1–2 (2017), 1–130.
[50] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 371–384.
[51] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: Data management for modern business applications. *ACM SIGMOD Record* 40, 4 (2012), 45–51.
[52] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA database–an architecture overview. *IEEE Data Engineering Bulletin* 35, 1 (2012), 28–33.
[53] Fork. 2020. Fork (system call). https://en.wikipedia.org/wiki/Fork_(system_call). Accessed in December 1, 2020.
[54] Peter Franaszek and John T. Robinson. 1985. Limitations of concurrency in transaction processing. *ACM Transactions on Database Systems (TODS)* 10, 1 (1985), 1–28.
[55] Craig Freedman, Erik Ismert, and Per-Ake Larson. 2014. Compilation in the Microsoft SQL server Hekaton engine. *IEEE Data Engineering Bulletin*. 37, 1 (2014), 22–30.
[56] Florian Funke, Alfons Kemper, Tobias Mühlbauer, Thomas Neumann, and Viktor Leis. 2014. HyPer beyond software: Exploiting modern hardware for main-memory database systems. *Datenbank-Spektrum* 14, 3 (2014), 173–181.
[57] Shen Gao, Jianliang Xu, Bingsheng He, Byron Choi, and Haibo Hu. 2011. PCMLogging: Reducing transaction logging overhead with PCM. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. 2401–2404.
[58] Hector Garcia-Molina and Kenneth Salem. 1992. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering* 4, 6 (1992), 509–516.
[59] Vishesh Garg, Abhimanyu Singh, and Jayant R. Haritsa. 2015. *On Improving Write Performance in PCM Databases*. Technical Report. Technical report, TR-2015-01, IISc.
[60] Dieter Gawlick and David Kinkade. 1985. Varieties of concurrency control in IMS/VS fast path. *IEEE Database Engineering Bulletin* 8, 2 (1985), 3–10.
[61] Goetz Graefe, Wey Guy, and Caetano Sauer. 2016. Instant recovery with write-ahead logging: Page repair, system restart, media restore, and system failover. *Synthesis Lectures on Data Management* 8, 2 (2016), 1–113.
[62] Goetz Graefe and Harumi Kuno. 2012. Definition, detection, and recovery of single-page failures, a fourth class of database failures. *Proceedings of the VLDB Endowment* 5, 7 (2012), 646–655.
[63] Goetz Graefe and William J. McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE, 209–218.
[64] Jim Gray and D. Bitton. 1988. Disk shadowing. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB'88), August 1988*, Vol. 88. 331–338.

[65] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. 173–182.
[66] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. 1981. The recovery manager of the system R database manager. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 223–242.
[67] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques*. Elsevier.
[68] Jim N. Gray, Raymond A. Lorie, and Gianfranco R. Putzolu. 1975. Granularity of locks in a shared data base. In *Proceedings of the 1st International Conference on Very Large Data Bases*. 428–451.
[69] Tandem Database Group. 1987. NonStop SQL: A distributed, high-performance, high-availability implementation of SQL. In *Proceedings of the High Performance Transaction Systems: 2nd International Workshop Asilomar Conference Center, Pacific Grove, CA, USA September 28–30*. Springer, 60–104.
[70] Le Gruenwald and Margaret H. Eich. 1991. *MMDB Reload Algorithms*. Vol. 20. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data (SIGMOD'91)*, April 1991 397-405.
[71] Le Gruenwald and Margaret H. Eich. 1994. MMDB reload concerns. *Information Sciences* 76, 1–2 (1994), 151–176.
[72] Wenming Guo and Zhiqiang Hu. 2010. Memory database index optimization. In *Proceedings of the 2010 International Conference on Computational Intelligence and Software Engineering*. IEEE, 1–3.
[73] Mohit Kumar Gupta, Vishal Verma, and Megha Singh Verma. 2014. In-memory database systems-a paradigm shift. *Arxiv Preprint Arxiv:1402.1258* (2014).
[74] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)* 15, 4 (1983), 287–317.
[75] Robert Hagmann. 1987. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. 155–162.
[76] Robert B. Hagmann. 1986. A crash recovery scheme for a memory-resident database system. *IEEE Trans. Comput.* 35, 9 (1986), 839–843.
[77] Theo Härder, Caetano Sauer, Goetz Graefe, and Wey Guy. 2015. Instant recovery with write-ahead logging. *Datenbank-Spektrum* 15, 3 (2015), 235–239.
[78] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2018. OLTP through the looking glass, and what we found there. In *Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker*. 409–439.
[79] Robin Harris. 2016. Windows leaps into the NVM revolution. https://www.zdnet.com/article/windows-leaps-into-the-nvm-revolution/. Accessed in July 22, 2020.
[80] Wytze Hazenberg and Sjoerd Hemminga. 2011. Main memory database systems: Opportunities and pitfalls. *SC@ RUG 2011 Proceedings* (2011), 113.
[81] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 289–300.
[82] Maristela Holanda, Angelo Brayner, and Sergio Fialho. 2008. A self-adaptable scheduler for synchronizing transactions in dynamically configurable environments. *Data & Knowledge Engineering* 66, 2 (2008), 223–242.
[83] Huiqi Hu, Xuan Zhou, Tao Zhu, Weining Qian, and Aoying Zhou. 2019. In-memory transaction processing: Efficiency and scalability considerations. *Knowledge and Information Systems* 61, 3 (2019), 1209–1240.
[84] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware logging in transaction systems. *Proceedings of the VLDB Endowment* 8, 4 (2014), 389–400.
[85] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. 1995. The clustra telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21st International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 469–477.
[86] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-atomic persistent memory updates via JUSTDO logging. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 427–442.
[87] H. V. Jagadish, Abraham Silberschatz, and S. Sudarshan. 1993. Recovering from main-memory lapses. In *Proceedings of VLDB*, Vol. 93. 391–404.
[88] Hosagrahar V. Jagadish, Daniel Lieuwen, Rajeev Rastogi, Abraham Silberschatz, and S. Sudarshan. 1994. Dali: A high performance main memory storage manager. In *Proceedings of VLDB*. 48–59.
[89] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. ZAB: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 245–256.
[90] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[91] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. 2014. Improving in-memory database index performance with intel® transactional synchronization extensions. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 476–487.

[92] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE)*. IEEE, 195–206.

[93] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 339–350.

[94] Hideaki Kimura. 2015. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 691–706.

[95] Hsiang-Tsung Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[96] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. 2013. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.* 36, 2 (2013), 6–13.

[97] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: The Works of Leslie Lamport*. 277–317.

[98] Leslie Lamport. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[99] Per-Ake Larson. 1988. Linear hashing with separatorsâa dynamic hashing scheme achieving one-access. *ACM Transactions on Database Systems (TODS)* 13, 3 (1988), 366–388.

[100] Per-Ake Larson and Justin Levandoski. 2016. Modern main-memory database systems. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1609–1610.

[101] Per-Ake Larson, Mike Zwilling, and Kevin Farlee. 2013. The Hekaton memory-optimized OLTP engine. *IEEE Data Engineering Bulletin*. 36, 2 (2013), 34–40.

[102] Juchang Lee, Yong Sik Kwon, Franz Färber, Michael Muehle, Chulwon Lee, Christian Bensberg, Joo Yeon Lee, Arthur H. Lee, and Wolfgang Lehner. 2013. SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 1165–1173.

[103] Tobin J. Lehman and Michael J. Carey. 1985. *A Study of Index Structures for Main Memory Database Management Systems*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[104] Tobin J. Lehman and Michael J. Carey. 1987. *A Recovery Algorithm for a High-performance Memory-resident Database System*. Vol. 16. Association for Computing Machinery (ACM).

[105] Tobin J. Lehman, Eugene J. Shekita, and L.-F. Cabrera. 1992. An evaluation of starburst's memory resident storage component. *IEEE Transactions on Knowledge and Data Engineering* 4, 6 (1992), 555–566.

[106] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 743–754.

[107] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Engineering Bulletin* 42, 1 (2019), 73–84.

[108] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

[109] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2014. Exploiting hardware transactional memory in main-memory databases. In *Proceedings of the 2014 IEEE 30th International Conference on Data Engineering*. IEEE, 580–591.

[110] Justin J. Levandoski, Per-Ake Larson, and Radu Stoica. 2013. Identifying hot and cold data in main-memory databases. In *Proceedings of the 2013 IEEE 29th International Conference onData Engineering (ICDE)*. IEEE, 26–37.

[111] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The bw-tree: A b-tree for new hardware platforms. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 302–313.

[112] Eliezer Levy and Abraham Silberschatz. 1992. Incremental recovery in main memory database systems. *IEEE Transactions on Knowledge and Data Engineering* 4, 6 (1992), 529–540.

[113] Kai Li and Jeffrey F. Naughton. 2000. Multiprocessor main memory transaction processing. In *Proceedings of the 1st International Symposium on Databases in Parallel and Distributed Systems*. IEEE Computer Society Press, 177–187.

[114] Liang Li, Guoren Wang, Gang Wu, and Ye Yuan. 2018. Consistent snapshot algorithms for in-memory database systems: Experiments and analysis. In *Proceedings of the 2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1284–1287.

[115] Liang Li, Guoren Wang, Gang Wu, Ye Yuan, Lei Chen, and Xiang Lian. 2018. A comparative study of consistent snapshot algorithms for main-memory database systems. *Arxiv Preprint Arxiv:1810.04915* (2018).

[116] Xi Li and Margaret H. Eich. 1993. Post-crash log processing for fuzzy checkpointing main memory databases. In *Proceedings of the 1993 9th International Conference on Data Engineering*. IEEE, 117–124.

[117] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *CIDR*.

[118] A.-P. Liedes and Antoni Wolski. 2006. Siren: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases. In *Proceedings of the 22nd International Conference on Data Engineering, 2006 (ICDE'06)*. IEEE, 99–99.

[119] Jun-Lin Lin and Margaret H. Dunham. 1997. A survey of distributed database checkpointing. *Distributed and Parallel Databases* 5, 3 (1997), 289–319.

[120] Barbara Liskov and Robert Scheifler. 1983. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 3 (1983), 381–404.

[121] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. 2012. Multi-version concurrency via timestamp range conflict management. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*. IEEE, 714–725.

[122] Lukas M. Maas, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. 2013. Buzzard: A numa-aware in-memory indexing system. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1285–1286.

[123] Sanjay Kumar Madria. 1997. A study of the concurrency control and recovery algorithms in nested transaction environment. *Comput. J.* 40, 10 (1997), 630–639.

[124] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. 2015. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1298–1309.

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

[126] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*. 183–196.

[127] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment* 11, 1 (2017), 1–13.

[128] Toshimi Minoura. 1983. *Multi-level Concurrency Control of a Database System*. Oregon State University, Computer Science Department.

[129] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided (RDMA) reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*. 103–114.

[130] C. Mohan. 1989. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*. Citesefer.

[131] C. Mohan. 1993. ARIES/LHS: A concurrency control and recovery method using write-ahead logging for linear hashing with separators. In *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE, 243–252.

[132] C. Mohan. 1999. Repeating history beyond ARIES. In *Proceedings of VLDB*, Vol. 99. 7–10.

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

[134] C. Mohan and Frank Levine. 1992. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. *ACM SIGMOD Record* 21, 2 (1992), 371–380.

[135] C. Mohan and Inderpal Narang. 1994. ARIES/CSA: A method for database recovery in client-server architectures. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. 55–66.

[136] C. Mohan and Hamid Pirahesh. 1991. ARIES-RRH: Restricted repeating of history in the ARIES transaction recovery method. In *Proceedings of the 7th International Conference on Data Engineering (1991)*. IEEE, 718–727.

[137] José Maria Monteiro, Angelo Brayner, and Sérgio Lifschitz. 2009. Using transaction isolation levels for ensuring replicated database consistency in mobile computing environments. In *Proceedings of the 8th ACM International Workshop on Data Engineering for Wireless and Mobile Access*. 1–8.

[138] Henrik Mühe, Alfons Kemper, and Thomas Neumann. 2011. How to efficiently snapshot transactional data: Hardware or software controlled? In *Proceedings of the 7th International Workshop on Data Management on New Hardware*. Association for Computing Machinery (ACM), 17–26.

[139] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2013. Instant loading for main memory databases. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1702–1713.

[140] MySQL. 2020. MySQL. http://www.mysql.com. Accessed in October 3, 2020.

[141] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[142] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.

[143] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. 1994. AlphaSort: A RISC machine sort. *ACM SIGMOD Record* 23, 2 (1994), 233–242.

[144] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 305–319.

[145] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 29–41.

[146] Oracle. 2020. Oracle | Integrated Cloud Applications and Platform Services. http://www.oracle.com. Accessed in December 4, 2020.

[147] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory management techniques for large-scale persistent-main-memory systems. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1166–1177.

[148] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. 2011. PLP: Page latch-free shared-everything OLTP. *Proceedings of the VLDB Endowment* 4, 10 (2011), 610–621.

[149] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-driving database management systems. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11 (CIDR'17)*.

[150] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage management in the NVRAM era. *Proceedings of the VLDB Endowment* 7, 2 (2013), 121–132.

[151] Peloton. 2019. Peloton - The Self-Driving Database Management System. https://pelotondb.io. Accessed in April 30, 2020.

[152] Danica Porobic, Erietta Liarou, Pınar Tözün, and Anastasia Ailamaki. 2014. ATraPos: Adaptive transaction processing on hardware islands. In *Proceedings of the 2014 IEEE 30th International Conference on Data Engineering*. IEEE, 688–699.

[153] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pınar Tözün, and Anastasia Ailamaki. 2012. OLTP on hardware islands. *Arxiv Preprint Arxiv:1208.0227* (2012).

[154] Philippe Pucheral, Jean-Marc Thévenin, and Patrick Valduriez. 1990. Efficient main memory data management using the DBGraph storage model.. In *Proceedings of the Symposium on VLDB*. 683–695.

[155] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database Management Systems*. McGraw-Hill.

[156] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. 2008. Constant-time query processing. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. IEEE, 60–69.

[157] Jun Rao and Kenneth A. Ross. 2000. Making B+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. 475–486.

[158] Simone Raoux, Geoffrey W. Burr, Matthew J. Breitwisch, Charles T. Rettner, Yi-Chou Chen, Robert M. Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, and Chung Hon Lam. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4.5 (2008), 465–479.

[159] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. 2016. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proceedings of the 2016 International Conference on Management of Data*. 1539–1551.

[160] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2012. Lightweight locking for main memory database systems. *Proceedings of the VLDB Endowment* 6, 2 (2012), 145–156.

[161] Ohad Rodeh. 2008. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)* 3, 4 (2008), 1–27.

[162] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. 1995. The impact of architectural trends on operating system performance. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 285–298.

[163] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.

[164] Kurt Rothermel and C. Mohan. 1989. *ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions*. IBM Thomas J. Watson Research Division.

[165] Yehoshua Sagiv. 1986. Concurrent operations on B∗-trees with overtaking. *Journal of Computer and System Sciences* 33, 2 (1986), 275–296.

[166] Kenneth Salem and Hector Garcia-Molina. 1989. Checkpointing memory-resident databases. In *Proceedings of the 5th International Conference on Data Engineering, 1989*. IEEE, 452–462.

[167] Kenneth Salem and Hector Garcia-Molina. 1990. System M: A transaction processing testbed for memory resident data. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 161–172.
[168] Caetano Sauer. 2017. *Modern Techniques for Transaction-oriented Database Recovery*. Ph.D. Dissertation. University of Kaiserslautern, Kaiserslautern, Germany. http://www.dr.hut-verlag.de/978-3-8439-3297-4.html.
[169] Caetano Sauer. 2019. *Modern Techniques for Transaction-oriented Database Recovery*. LNI, Vol. P-289. Gesellschaft für Informatik, Bonn. https://doi.org/10.18420/btw2019-30
[170] Caetano Sauer, Goetz Graefe, and Theo Härder. 2015. Single-pass restore after a media failure. In *Proceedings of the Datenbanksysteme für Business, Technologie und Web (BTW 2015)* (2015).
[171] Caetano Sauer, Goetz Graefe, and Theo Härder. 2017. Instant restore after a media failure. In *Proceedings of the European Conference on Advances in Databases and Information Systems*. Springer, 311–325.
[172] Caetano Sauer, Goetz Graefe, and Theo Härder. 2018. FineLine: Log-structured transactional storage and recovery. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2249–2262.
[173] Bianca Schroeder and Garth A. Gibson. 2007. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, Vol. 78. IOP Publishing, 012022.
[174] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. 2015. NVM MALLOC: Memory allocation for NVRAM. *ADMS@ VLDB* 15 (2015), 61–72.
[175] David Schwalb, Girish Kumar BK, Markus Dreseler, S. Anusha, Martin Faust, Adolf Hohl, Tim Berning, Gaurav Makkar, Hasso Plattner, and Parag Deshmukh. 2016. Hyrise-NV: Instant recovery for in-memory databases using non-volatile memory. In *International Conference on Database Systems for Advanced Applications*. Springer, 267–282.
[176] David Schwalb, Martin Faust, Markus Dreseler, Pedro Flemming, and Hasso Plattner. 2016. Leveraging non-volatile memory for instant restarts of in-memory database systems. In *Proceedings of the 2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 1386–1389.
[177] Dennis Shasha. 1985. What good are concurrent search structure algorithms for databases anyway? *IEEE Database Engioneering Bulletin*. 8, 2 (1985), 84–90.
[178] Dennis Shasha and Nathan Goodman. 1988. Concurrent search structure algorithms. *ACM Transactions on Database Systems (TODS)* 13, 1 (1988), 53–90.
[179] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: The end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery (ACM), 731–742.
[180] Michael Stonebraker. 1991. Managing persistent objects in a multi-level store. *ACM SIGMOD Record* 20, 2 (1991), 2–11.
[181] Michael Stonebraker and Ariel Weisberg. 2013. The VOLTDB main memory DBMS. *IEEE Data Engineering Bulletin* 36, 2 (2013), 21–27.
[182] Jimmy P. Strickland, Peter P. Uhrowczik, and Vern L. Watts. 1982. IMS/VS: An evolving system. *IBM Systems Journal* 21, 4 (1982), 490–510.
[183] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. 2008. The missing memristor found. *Nature* 453, 7191 (2008), 80–83.
[184] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.
[185] Kian-Lee Tan, Qingchao Cai, Beng Chin Ooi, Weng-Fai Wong, Chang Yao, and Hao Zhang. 2015. In-memory databases: Challenges and opportunities from software and hardware perspectives. *ACM SIGMOD Record* 44, 2 (2015), 35–40.
[186] Yong Chiang Tay, Nathan Goodman, and Rajan Suri. 1985. Locking performance in centralized databases. *ACM Transactions on Database Systems (TODS)* 10, 4 (1985), 415–462.
[187] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 1–12.
[188] TimesTen. 2020. Oracle TimesTen In-Memory Database Documentation. https://docs.oracle.com/database/timesten-18.1/. Accessed in December 1, 2020.
[189] CORPORATE TimesTen Team. 1999. In-memory data management for consumer transactions the TimesTen approach. In *ACM SIGMOD Record*, Vol. 28. Association for Computing Machinery (ACM), 528–529.
[190] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery (ACM), 18–32.
[191] Allen B. Tucker. 2004. *Computer Science Handbook*. CRC Press.

[192] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*. 1541–1555.

[193] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent memory I/O primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–7.

[194] Marcos Vaz Salles, Tuan Cao, Benjamin Sowell, Alan Demers, Johannes Gehrke, Christoph Koch, and Walker White. 2009. An evaluation of checkpoint recovery for massively multiplayer online games. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1258–1269.

[195] Tianzheng Wang and Ryan Johnson. 2014. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment* 7, 10 (2014), 865–876.

[196] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. 2014. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th European Conference on Computer Systems*. 1–15.

[197] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Elsevier.

[198] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. 2000. Database replication techniques: A three parameter classification. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS-2000)*. IEEE, 206–215.

[199] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing database column scans with complex predicates.. In *ADMS@ VLDB*. 1–12.

[200] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment* 2, 1 (2009), 385–394.

[201] Yingjun Wu, Wentian Guo, Chee-Yong Chan, and Kian-Lee Tan. 2017. Fast failure recovery for main-memory DBMSs on multicores. In *Proceedings of the 2017 ACM International Conference on Management of Data*. Association for Computing Machinery (ACM), 267–281.

[202] Tang Yanjun and Luo Wen-hua. 2010. A model of crash recovery in main memory database. In *Proceedings of the 2010 International Conference on Computer Design and Applications (ICCDA)*, Vol. 5. IEEE, V5–206.

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

[204] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment* 8, 3 (2014), 209–220.

[205] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking database high availability with RDMA networks. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1637–1650.

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

[207] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. MOJIM: A reliable and highly-available non-volatile memory system. In *ACM SIGARCH Computer Architecture News*, Vol. 43. Association for Computing Machinery (ACM), 3–18.

[208] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of OSDI*, Vol. 14. 465–477.

Received August 2020; revised December 2020; accepted December 2020