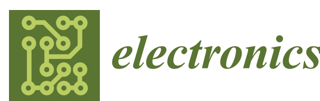


Article

FIR: Achieving High Throughput and Fast Recovery in a Non-Volatile Memory Online Transaction Processing Engine

Jianhao Wei *, Qian Zhang, Yiwen Xiang and Xueqing Gong

Software Engineering Institute, East China Normal University, Shanghai 200062, China; 52184501012@stu.ecnu.edu.cn (Q.Z.); 51215902109@stu.ecnu.edu.cn (Y.X.); xqgong@sei.ecnu.edu.cn (X.G.)
* Correspondence: 52215902007@stu.ecnu.edu.cn

Abstract: Existing databases supporting Online Transaction Processing (OLTP) workloads based on non-volatile memory (NVM) have not fully leveraged hardware characteristics, resulting in an imbalance between throughput and recovery performance. In this paper, we conclude with the reason why existing designs fail to achieve both: placing indexes on NVM results in numerous random writes and write amplification for index updates, leading to a decrease in system performance. Placing indexes on dynamic random access memory (DRAM) results in much time consumption for rebuilding indexes during recovery. To address this issue, we propose FIR, an NVM OLTP Engine with the fast rebuilding of the DRAM indexes, achieving instant system recovery while maintaining high throughput. Firstly, we design an index checkpoint strategy. During recovery, the indexes are quickly rebuilt by the bottom-up algorithm with index checkpoints. Then, to achieve instant recovery of the entire engine after rebuilding indexes, we optimize the existing log-free design by leveraging time-ordered storage, which significantly reduces the number of NVM writes. We also implement garbage collection based on data redistribution, enhancing system availability. The experimental results demonstrate that FIR achieves 98% of the performance of state-of-the-art OLTP Engine when running TPCC and YCSB. And the recovery speed of FIR is 43.6×–54.5× faster, achieving near-instantaneous recovery.

Keywords: non-volatile memory; OLTP Engine; index checkpoint; recovery; performance

Academic Editor: Hyungjin Kim

Received: 21 October 2024
Revised: 21 November 2024
Accepted: 26 November 2024
Published: 26 December 2024

Citation: Wei, J.; Zhang, Q.; Xiang, Y.; Gong, X. FIR: Achieving High Throughput and Fast Recovery in a Non-Volatile Memory Online Transaction Processing Engine. *Electronics* **2025**, *14*, 39. https://doi.org/10.3390/electronics14010039

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Dynamic random access memory (DRAM) databases that support OLTP workloads can typically process millions of transactions per second [1–7]. Their characteristics of high throughput and low latency are increasingly favored by a growing number of users. However, the shortcomings of DRAM are evident, including expensive per-unit costs; high power consumption, leading to elevated operational expenses; limitations on the data scale due to capacity constraints; and a risk of data loss in the event that the power is off.

Non-volatile memory (NVM) features byte-level addressing, non-volatile storage, DRAM-like speed, and latency, and it can scale in capacity up to 6 TB [8,9]. Researchers have been actively exploring the possibility of constructing an OLTP Engine with NVM, including NV-Logging [10], FOEDUS [11], WBL [12], Zen [13], Hyrise [14], and Falcon [15]. The advantages of using NVM to build an OLTP Engine are evident. In ideal scenarios, an NVM OLTP Engine can achieve throughput and much faster recovery speeds compared to a DRAM OLTP Engine. However, there are still some performance gaps between the current NVM and DRAM, including the following: (1) the minimum read/write size of NVM is 256B, while DRAM's is only 64B; (2) the bandwidth of NVM is only 1/3 to

1/2 of that of DRAM; and (3) the read performance of NVM is higher than the write performance [8,9,16–19]. Therefore, there are limitations in designing NVM OLTP Engines. Similar research directions also encompass the NVM Index [20–24], Data Placement on NVM [25–27], and Logging and Recovery [10,12,28–34].

In this paper, we focus on OLTP Engines for NVM, including their throughput and recovery speed. Firstly, we classify the data in an OLTP Engine and summarize two designs of NVM OLTP Engines: the NVM-based OLTP Engine and the DRAM–NVM Hybrid OLTP Engine. Then, we discover that these two designs cannot achieve both fast recovery and high throughput. Following our experiments and analysis, we present our insights: (1) Placing indexes on NVM results in a large number of random writes and write amplifications for index updates, leading to a decrease in system performance. (2) Placing the index in DRAM results in much time consumption for rebuilding indexes during recovery.

Based on the above insights, we propose FIR, an NVM OLTP Engine featuring the fast rebuilding of the DRAM index, achieving instant system recovery while maintaining high runtime throughput. First, by utilizing a timestamp-based path-marking algorithm, we mark all index nodes modified by transactions, which can lead to obtaining all modified leaf nodes as incremental index checkpoint data during the flushing checkpoint. Then, the system utilizes a time-cost model to dynamically choose between incremental and full checkpoints at the runtime. Additionally, we use time-ordered storage to strictly control the persistent ordering of tuple versions in NVM. On this basis, we design a redo/undo algorithm for recovery that does not require logging. The longer the system runs before a crash, the closer the time complexity of the algorithm approaches to $O(1)$.

In summary, on the one hand, FIR can rapidly recover indexes in DRAM, performing redo/undo operations quickly, enabling instant recovery. On the other hand, FIR does not need to write logs, execute transactions in a multi-thread-friendly manner, or access indexes in DRAM, which ensures high throughput.

The contributions of this paper are as follows.

1. We analyze the impact of placing the index in DRAM or NVM on the NVM OLTP Engine through experiments.
2. We propose a mechanism for the fast recovery of the DRAM index in the NVM OLTP Engine. Additionally, we implement FIR, an NVM OLTP Engine that can maintain a high throughput and recovery speed.
3. The experimental results demonstrate that FIR achieves 98% of the performance of the best OLTP Engine when running TPCC and YCSB, and the recovery speed of FIR is 43.6×–54.5× faster, achieving near-instant recovery.

2. Background and Motivation

In this section, we first discuss the types of data in OLTP Engines, as well as two strategies for recovery: recovery with a transaction result and recovery with a transaction state. Next, we introduce the designs of two different NVM OLTP Engines and their respective typical systems. Following that, we present the concept of index checkpoints and then introduce our insights and challenges.

2.1. Data Persistence and Recovery in OLTP Engines

There are various types of data in OLTP Engines, which have different requirements for persistence. We group the data into five categories based on their usage (see Table 1): *Content Data*, *Metadata*, *Indexes*, *Runtime Data*, and *Recovery Data*. *Content Data* refers to user and system data managed by the OLTP Engine, which are usually stored in tables in the form of a single version or multiple versions. *Metadata* are the internal data used for layout and version management, such as version timestamps in tuple version headers. *Indexes* are

the access paths to the *Content Data*, such as B^+-tree. It is not necessary to keep these three types of data in non-volatile storage devices. They can be restored through backups and logs when the system crashes or restarts. *Runtime Data* include various buffers, lock tables, active transaction tables, and other temporary data used for caching and concurrency control when the engine is running. They do not need to be restored when the engine crashes or restarts. Therefore, there is no keep to keep *Runtime Data* in non-volatile storage devices. The content of the *Recovery Data* is the result of transactions such as redo log or the states of active transactions such as active transaction table. They are always kept in non-volatile storage devices, as they are used to restore the engine to a consistent state in the event of a transaction rollback or crash.

Table 1. Persistence requirements of different types of data in an OLTP Engine.

Data Type	Description	Data Structure	Persistence
Content Data	User data and system data stored in tables	Tuple Version Store, etc.	Not necessary
Metadata	Internal data used for layout and version management	Tuple Version Header, Block Header, etc.	Not necessary
Indexes	Access paths to user data	Hash Index, B^+-Tree, etc.	Not necessary
Runtime Data	Temporary Data used when the engine is running	Buffers, Active Transaction Table, etc.	No need
Recovery Data	Data used to ensure transaction atomicity and durability	Log Entries, Checkpoint, etc.	Always

The recovery strategy used by an OLTP Engine is determined by the content of its recovery data. For systems where its recovery data contain the results of transactions, they can be restored to a consistent state by redo and undo processing. We take it as *Result-based Recovery (RR)*. The most typical example is Write-Ahead Logging (WAL) [35] and its derivatives [11,36–43]. In such systems, *Content Data*, *Metadata*, and *Indexes* do not need to be kept, but most systems keep them for performance. For systems where its recovery data only contain the states of active transactions, *Content Data* and *Metadata* must be kept, including the data of uncommitted transactions. The result of uncommitted transactions will be discarded during the recovery processing. We take it as *State-based Recovery (SR)*. In such systems, *Recovery Data* can be stored with *Metadata*, such as Log-Free [13] and Log-as-Data [44,45], or stored separately, such as Write-Behind Logging (WBL).

2.2. The Designs of NVM OLTP Engines

We have summarized two designs of NVM OLTP Engines, depending on whether the first three types of data are persistent and the type of persistent storage devices. We primarily focus on recovery strategies and their impact on system throughput.

Design 1 is called an NVM-based OLTP Engine, which stores the first three types of data in NVM, as depicted in Figure 1a. During recovery, it employs the SR strategy, which rapidly undoes uncommitted transactions using transaction state messages out of tuple versions. WBL, NV-Logging, and Hyrise adopt this design, with the most representative being WBL. The specific strategy is as follows: first, WBL ensures that all write operations to NVM have been completed before transaction submission. At the same time, transactions adopt a group commit approach, recording a log entry called Write-Behind Log (WBL) upon their commit. This log records the transaction ID of the most recently committed transaction and the maximum transaction ID for the next group commit. During recovery, the system

can be restarted by analyzing the logs and recording the transaction ID of the most recently committed transaction, because index access will skip the uncommitted versions based on this transaction ID. At the same time, the garbage collection mechanism begins to work, rolling back the effects of transactions between commit timestamps and cleaning up outdated WBL records. Although Design 1 can achieve instant recovery, it requires the indexes to be stored in NVM. Accessing the indexes leads to many NVM-unfriendly operations, such as small writes, slowing down the entire system.

Design 2 is called the DRAM-NVM Hybrid OLTP Engine. This strategy keeps only the content data and partial metadata in NVM while storing the index and other metadata in DRAM, as depicted in Figure 1b. Although it is log-free, the transaction state messages that should be in logs are now stored in-line with tuple versions, such as transaction ID and commit timestamps. During recovery, it employs the SR strategy, which scans all tuple versions to obtain transaction state messages and builds the index in DRAM. Zen, Falcon, and Tairpmem [44] adopt this design, and one of the most representative ones is Zen. The specific strategy is as follows: Zen places indexes in DRAM, with only tuple versions and some necessary metadata in NVM. This approach reduces a significant amount of NVM read and write operations compared to WBL and Hyrise. Furthermore, to fully utilize DRAM and enhance system performance, Zen also maintains a tuple-level caching mechanism in DRAM, Met-Cache. This mechanism uses eviction algorithms to keep frequently accessed data resident in the cache, while infrequently accessed data are swapped to the NVM-Tuple Heap. Moreover, to further improve multi-core performance, Zen divides its Met-Cache into multiple equal-sized regions, one per transaction processing thread. In terms of recovery, Zen adopts a log-free mode [13]. It adds three additional attributes, LP (last persisted), Tx-CTS (transaction ID), and Deleted (whether the tuple version is obsolete), to each tuple version in NVM to manage data persistence. During recovery, the system uses an algorithm with a time complexity of $O(nlog(n))$ to scan the entire table for data rollback, clear obsolete versions, and rebuild indexes in DRAM.

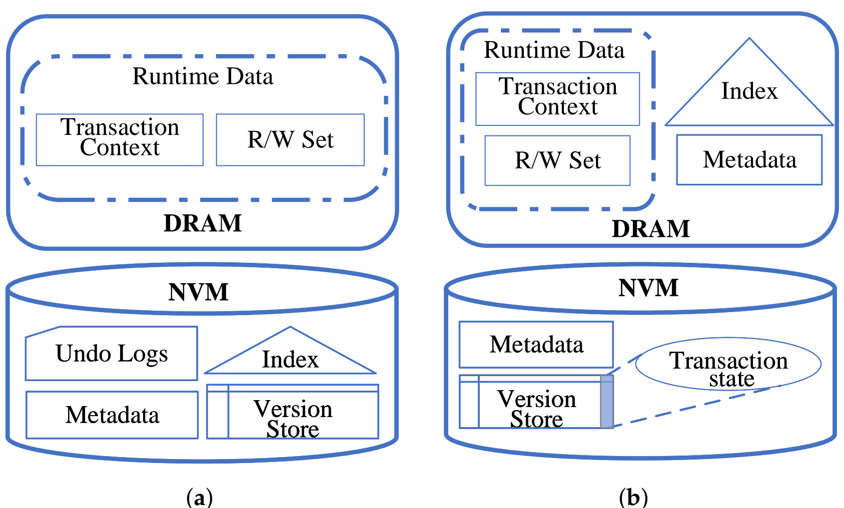


Figure 1. Two designs of the NVM OLTP Engine. (**a**) NVM-based; (**b**) DRAM–NVM hybrid.

Although Zen avoids a significant amount of small writes and seemingly improves the system performance through caching mechanisms, it still has several issues. In terms of the system performance, the design of met-cache seems somewhat redundant. The core principle of database cache design is to take advantage of the significant difference in read performance between DRAM and persistent storage. Preloading data that may be read into DRAM and asynchronously writing the data to persistent storage can accelerate read and write operations. However, the read performances of NVM and DRAM are similar, and

once a tuple version is modified, it must be written to met-cache first and stored in NVM immediately because Zen must ensure that committed tuple versions exist in NVM. When there are numerous modification operations, the write pressure on both NVM and DRAM becomes significant, leading to a decrease in system performance. In terms of recovery, Zen needs to spend a significant amount of time scanning all data to eliminate the effects of uncommitted transactions and also needs to build indexes in DRAM.

Designs 1 and 2 are the main designs for current NVM OLTP Engines. The former has a better recovery speed but suffers from low throughput compared to Design 2 [12,13]. The latter has a higher throughput but has poor performance in recovery compared to Design 1. The persistence of the index is the fundamental factor contributing to the observed phenomena. Firstly, when the index is in NVM, there is no need to rebuild the index during a system crash; only transaction states are enough for a fast recovery. However, during the system runtime, there are many read and write operations on the index, and the size of data for these operations is often less than 256B (minimum read/write unit of NVM), leading to severe NVM write amplification. Secondly, the NVM write speed is much slower than DRAM, significantly slowing down the system throughput [9,16,17].

2.3. Index Checkpoint

According to Design 2, since rebuilding the index requires a significant amount of time, if it is possible to keep the DRAM index as a checkpoint, it can ensure system performance and quickly load the index during recovery. In 2022, scholars first introduced the concept of DRAM-DB index checkpoints [46], which periodically store an index snapshot at a certain moment as a checkpoint on persistent devices to reconstruct it faster during recovery. They proposed IACoW, as depicted in Figure 2. This strategy maintains a relationship array from node IDs to nodes for all nodes in the index tree, and the index records no longer indicate the position of child nodes but rather their IDs in the array. Therefore, any node access is performed through this array. When a checkpoint begins, the checkpoint thread sequentially scans all nodes in the array and keeps their corresponding nodes. To address the issue of index consistency, IACoW retains the index versions from the previous checkpoint to the current checkpoint, ensuring that transactions can access consistent versions. Although IACoW appears to be excellent in DRAM-DB, we believe that this solution is not suitable for NVM-DB. Firstly, this design incurs significant additional performance overhead, such as managing the node array and garbage collection of obsolete versions. Secondly, keeping such massive index checkpoint data is time-consuming. These issues are acceptable in DARM-DB, where runtime performance bottlenecks primarily stem from the persistence of data checkpoints and log writes. Additionally, even if IACoW can enable instant recovery of the index, loading data checkpoints and replaying logs themselves take a considerable amount of time, so achieving instant recovery for the entire system is impractical when using index checkpoints in DRAM-DB. In Section 4, we implement IACoW in NVM-DB, test its throughput and recovery performance, and demonstrate our point.

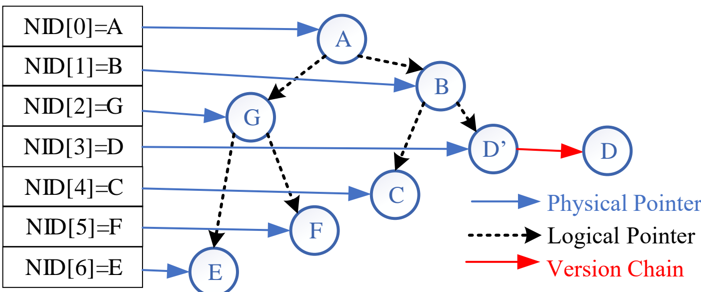


Figure 2. DRAM-DB index checkpoint: IACoW.

2.4. Challenge

Figure 3a shows the performance of DRAM indexes in both NVM and DRAM, including Masstree [47], B-Tree [48], BW-Tree [49], ART [50], and CSB [51]. Figure 3b displays the performance comparison between the DRAM index (Masstree) and NVM indexes specifically designed for NVM, including Hyrise, Roart [23], Nbtree [21], and wB+-Trees [24]. We use 20 threads to perform 1 billion random insert operations and record the throughput per second, with the initial index size being 100 million entries and a key size of 64 bits. It can be seen that simply executing the DRAM index in NVM results in a performance drop of 45% to 63%. However, even the indexes designed for NVM cannot reach the performance of the DRAM index, achieving only 51% to 78% of its performance. Based on the experiments and analysis mentioned above, it is evident that to improve the throughput of the NVM OLTP Engine, indexes should be placed in DRAM. However, once the system crashes, a significant amount of time is required for reconstruction. If the goal is to enhance recovery speed, then indexes need to be persisted, but this may lead to a loss in runtime performance. Therefore, instead of designing an NVM index, we design a mechanism for quickly recovering indexes in DRAM, allowing access to indexes in DRAM and enabling fast recovery.

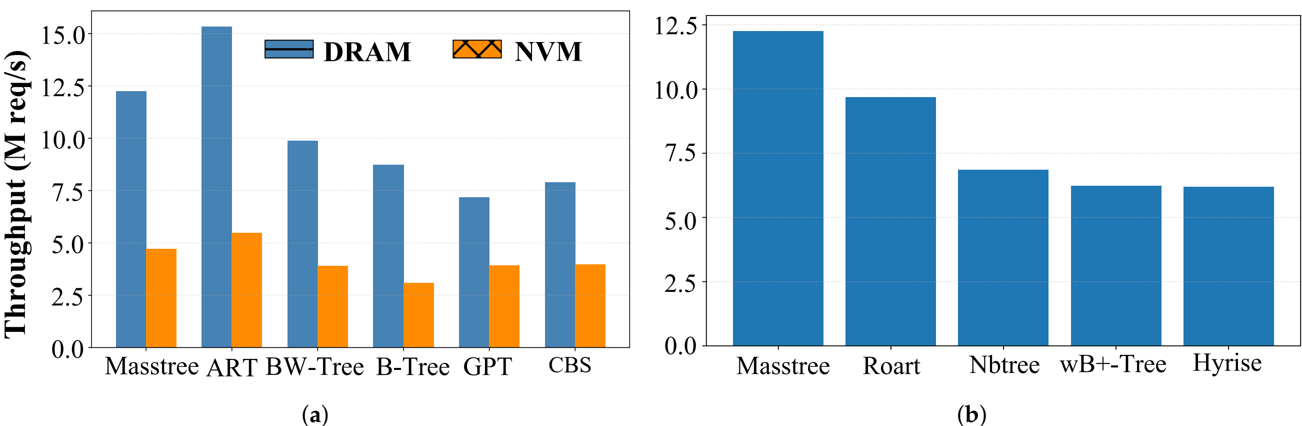


Figure 3. Performance of DRAM Indexes and NVM Indexes. (**a**) Throughput of DRAM indexes inserting into DRAM and NVM; (**b**) throughput of NVM indexes inserting into NVM.

3. Related Work

3.1. OLTP Engines for NVM

Besides the previously mentioned WBL and Zen, there have been several OLTP Engines based on NVM proposed before this. FOEDUS stores tuple version in snapshot pages in NVM using page caching in DRAM. FOEDUS maintains indexes in DRAM and executes transactions. The redo logs generated by transactions are periodically maintained by background threads, and new snapshots in NVM are generated through computations similar to Map-Reduce [52]. For recovery, FOEDUS uses redo logs to recover committed cached data in DRAM. Falcon optimizes for eADR-enabled NVM [53], allowing cache persistence and updating data through in-place updates on eADR-enabled NVM. Redo logs are recorded in the persist cache to reduce NVM writes, and the index is stored in NVM for fast recovery. In terms of transaction processing, Falcon stores version information in DRAM to support multi-version concurrency control. Unfortunately, the eviction of the persist cache is managed by the Operating System, and Falcon cannot guarantee that redo logs always reside in it. Once the system is under a heavy load, the swapping in and out of the persist cache will inevitably increase, leading to unnecessary NVM writes.

3.2. Index Checkpoint for DRAM-Based Engine

In previous works, most researchers focus on checkpoints for content data in DRAM-based Engine, such as VoltDB [3], SiloR [54], COU [55], Zigzag [56], and HyPer [57]. If the recovery relies on redo logs only, it takes too much time. Therefore, researchers store snapshots of the versions as checkpoints on persistent devices. The study [46] first proposed the concept of index checkpoints. The paper points out the necessity of index checkpoints for recovery and designs a multi-version index for checkpoints. During recovery, it not only rebuilds the indexes based on full index checkpoints but also loads version checkpoints to recover the tuple heap. In contrast to [46], FIR does not need to recover the tuple heap and uses index leaf nodes as checkpoints, eliminating the need to maintain a large size of the multi-version index.

3.3. Recovery with NVM

Many previous works leverage the fast and persistent feature of NVM to accelerate recovery speed and optimize overall system throughput. NV-Logging places the log buffer in NVM and the data in DRAM, avoiding the more expensive software-based I/O interface access. Reference [34] uses NVM and distributed logs on multi-core and multi-socket hardware. During the redo process of recovery, modifications to different pages can be replayed in parallel, while undo is parallelized at the transaction level. Reference [58] proposes a log record method called Proteus, which controls its log region through two new log instructions. It can drop log updates that have not yet been written back to NVM by the time a transaction is considered durable. Silo [45] temporarily maintains the undo and redo logs on the chip during transaction execution. After the transaction commits, Silo leverages the new data in these on-chip logs to update the NVM data region in place instead of conservatively writing logs as useless backups in common cases where no crash occurs.

4. FIR Design

In this section, we first present the storage of FIR and its internal modules (Section 4.1). Subsequently, we describe the transaction processing (Section 4.2). Then, we explain the recovery strategy of the entire system, including the core index checkpoint technology (Section 4.3). Finally, we introduce the garbage collection strategy (Section 4.4).

4.1. Time-Ordered Storage

Figure 4 illustrates the design of FIR. Each table is logically partitioned into several regions, where tuple content data from the same table within the same region are stored in NVM on a block-by-block basis. Each block has its corresponding tile in DRAM, storing the majority of metadata used for transaction processing. The *table_region_addr* records the mapping relationship between the table and the region, as well as the last block and tile write position of every region. Additionally, each table corresponds to at least one index in DRAM. In the index checkpoint module, each index corresponds to one index checkpoint, a snapshot of the index leaf nodes at a specific time. The index checkpoints are stored in NVM and managed by the index checkpoint thread, facilitating the rapid construction of indexes during recovery.

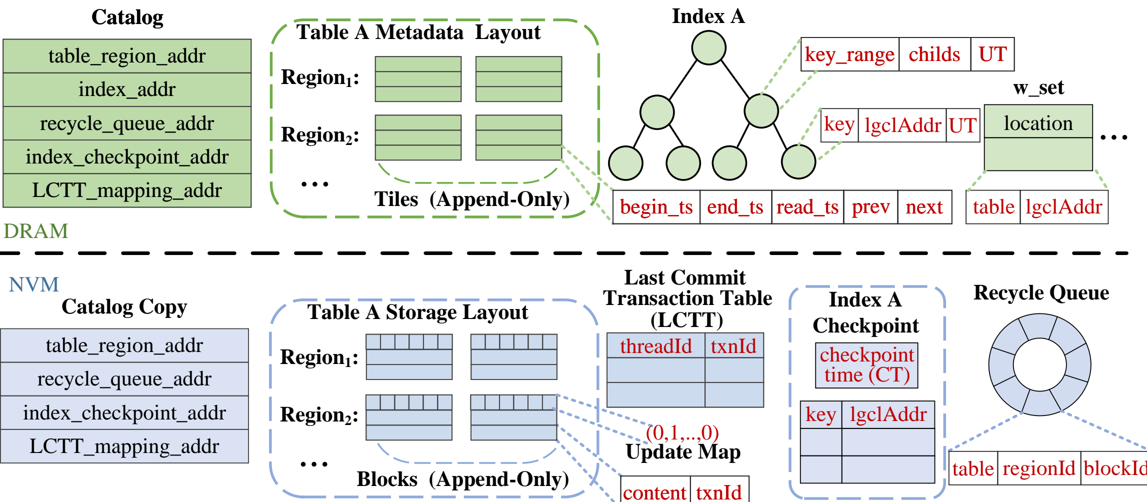


Figure 4. FIR architecture.

Data Layout. In FIR, a *tuple* has a unique *ID* and multiple tuple versions that are stored as a tuple heap. Each tuple version consists of two parts: the *tuple_header* in DRAM and the *tuple_content* in NVM. Each *tuple_content* includes content data and a *txnId* of 64 bits representing the transaction generating the *tuple_content*. Every 2048th *tuple_content* forms a *block*. Each block maintains one *update_map*, a bitmap with a length of 256 bytes (2048 bits), aligned to the write size of NVM, where each bit corresponds to the position of a *tuple_content* in the *block*, and the value indicates whether the tuple version is obsolete. The *tuple_header* is the metadata for each tuple version, including *begin_ts*, *end_ts*, and *read_ts* for concurrency control; *next* pointing to the next version of tuple; and *prev* pointing to the previous *tuple_version*. Every 2048th *tuple_header* forms a *tile*.

Index. FIR supports various types of tree-structured indexes, where the index key is the tuple ID or other fields, and the index value (*lgclAddr*) is a 64-bit pointer, pointing to the logical address of the latest version of the tuple. The logical address of a tuple version is a 64-bit integer. The first 16 bits represent the *regionId*, the middle 32 bits represent the *tileId*(*blockId*), and the last 16 bits represent the offset of the record in the block. Every node in the index maintains a field (Update Time) UT of 64 bits, representing the transaction that executes the most recent update.

Index Checkpoint. Each index corresponds to one index checkpoint, which is generated and flushed to NVM periodically. Each checkpoint record consists of the index keys and values of a leaf node entry. There is a 64-bit field called *CT* that represents the start time of the current checkpoint. Additionally, we design two checkpoint modes: full and incremental. The former consists of all leaf nodes of the tree, while the latter consists of modified or inserted leaf nodes within a specific time period. Furthermore, we design a time cost model, which is used to calculate the time cost of incremental and full checkpoints, thereby determining which type of checkpoint to use.

Last Commit Transaction Table. The Last Commit Transaction Table (LCTT) is an array of ThreadId and txnId that stores the ID of the most recent committed transaction for each thread. We keep the writes to LCTT aligned with NVM.

Recycle Queue. The Recycle Queue is a circular collection queue for garbage collection that stores the logical addresses of blocks (tiles) awaiting recycling.

Write Set. The Write Set (*w_set*) is an array of table names and logical addresses in DRAM, recording the locations of new versions updated or inserted by a transaction. Each transaction has its own Write Set, which is used for conflict detection and transaction rollback.

We control all write operations in an "Append Only" manner by implementing several key mechanisms. First, the logical partitioning of storage divides each table into multiple regions, where write operations within a region are strictly appended to the end of the existing data structures. This logical partitioning ensures that writes are sequential and localized, reducing fragmentation and improving write performance. Each thread is bound to a region, and each transaction is assigned to a single thread, ensuring that multiple writes of the same transaction occur only within the corresponding region and in an "Append Only" manner. Second, tuple content data are stored in a time-ordered tuple structure, where each new tuple version is appended as a new entry, guaranteeing that all tuple versions within a specific table and region are arranged in chronological order. Each tuple version is assigned a unique timestamp, allowing the system to easily track the order of changes. Moreover, once a block is written to NVM, it is treated as immutable, meaning that no in-place updates or deletions occur; instead, any changes result in the creation of new blocks, thus avoiding the complexities associated with modifying existing data and simplifying the concurrency control. Lastly, each tuple has a unique ID and can have multiple versions, with each version containing a tuple header in DRAM and the corresponding tuple content in NVM. The tuple header includes metadata, such as timestamps and pointers to the previous and next versions, enabling efficient access and management of tuple versions. Through these mechanisms, we ensure the efficiency of write operations and the integrity of the data.

This approach has several advantages. Firstly, placing the index and most metadata in DRAM significantly reduces the write pressure on NVM. Secondly, it allows FIR to ensure data consistency without the need for any logs; during recovery, redo/undo operations can be performed without scanning the entire table because the data themselves serve as a log and are organized in temporal order. Thirdly, it avoids the additional management overhead caused by a large number of data fragments (obsolete versions) within blocks; refer to Section 4.4 for specific garbage collection strategies.

4.2. Transaction Processing

4.2.1. Transaction Lifetime

FIR employs MVTO (Multi-Version Timestamp Ordering) as the concurrency control protocol to manage transactions. Each transaction is assigned to a working thread for execution, and each working thread can only write within its own region. Assuming that transaction Txn 200, which needs to update a tuple in Table A, is assigned to working thread 1 corresponding to $Region_1$, its lifetime is shown in Figure 5. First of all, the working thread starts with a read operation (Steps 1 to 4). It obtains the logical address of the tuple through the index and updates the UT of nodes in the index access path to the current txnId (details in Section 4.3). Then, it checks for transaction conflicts; if there are none, it acquires the write lock for the tuple. Subsequently, it accesses the *tuple_header* in the tile using this address and updates *end_ts* with *txnId*. Following that, it sets the value of the *update_map* corresponding to the old version to 1 and obtains the tuple version content. Second, the working thread writes a new tuple version into table A (Steps 5 to 7). It obtains an empty slot from the last tile, and the logical address of this tuple version is the region ID, tile number, and slot offset. After that, the working thread writes a new *tuple_header* into the slot, setting *begin_ts* to *txnId*, *prev* to the previous version, and the previous version's *next* to the lastest version, and its adds table name A and the logical address of the latest version to the *w_set* (the write set). Since the *tuple_header* in the tile corresponds one-to-one with the tuple version in the block, their logical addresses *(tableId, regionId, offset)* are the same. At the same time, it lets the pointer of the index value point to this logical address.

In this way, accessing the latest version can be performed without modifying the value of the index.

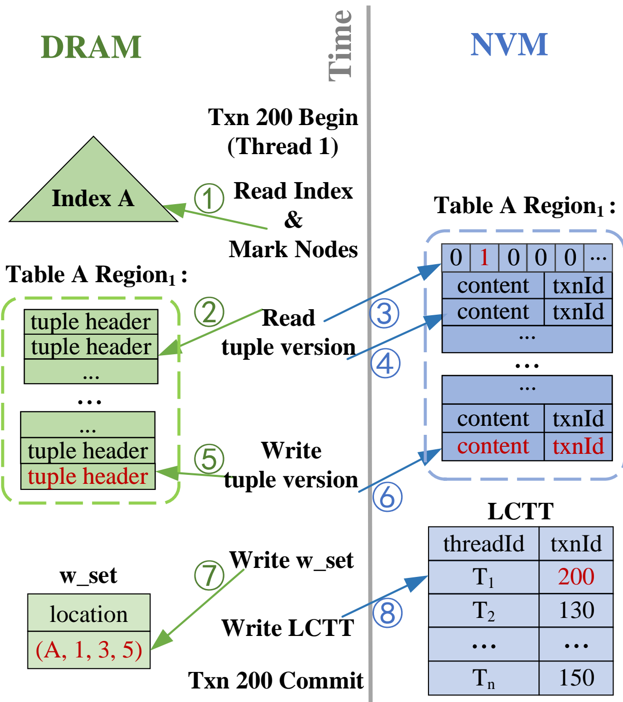


Figure 5. Lifetime of a transaction.

Following this, the tuple content persists in the NVM. To ensure correctness and minimize the potential for write amplification, FIR adopts a strategy using *clwb* and *sfence*. The entire persistence process is outlined in Algorithm 1. Initially, the working thread obtains the persistence location (Line 3) from the *table_region_addr* using *lgclAddr* (logical address). Then, it uses *memcpy* to copy the data from *newtups* into *persistTups* sequentially (Lines 4 to 7). Finally, the *clwb* and *sfence* instructions are used to flush 64 bits from the cache to the WPQ (Write Pending Queue) until completion (Lines 8 to 10). It is worth noting that, since the multiple tuple versions generated by one update operation are contiguous on NVM, we choose to perform *sfence* immediately after each *clwb*, ensuring that the data can be evicted from the cache in order. This effectively enhances the efficiency of NVM flushing and reduces the write amplification.

Algorithm 1: Persist Function

```
1  Function Persist(newTups[], persistTups, lgclAddr)
2      tupleLen ← sizeof(int64_t);
       // Obtain persistence location
3      persistTups ← GetPstLoc(lgclAddr);
4      for i ← 0 to newTups.len − 1 do
           // Prepare data
5          tupleData ← newTups[i];
6          memcpy(persistTups, &tupleData, tupleLen);
7          persistTups ← persistTups + tupleLen;
8      for i ← 0 to newTups.len × tupleLen − 1 step CACHE_SIZE do
           // Persist tuple
9          clwb(persistTups + i);
10         sfence;
```

Finally, during the transaction commit phase (Step 8), the working thread iterates through the *write_set*, releasing the write lock of each tuple version. Then, it places the current working thread and *txnId* into the group commit queue and updates the *txnId* associated with the working thread in the LCTT. If the transaction aborts, the working thread iterates through the *write_set*, resetting the last recent empty slot positions of blocks corresponding to the tuple versions to the state before the transaction started. In the above process, there are a large number of small write operations to the *tuple_header*. We place these operations in DRAM to reduce NVM writes.

4.2.2. Support for Long Transactions

When an active transaction exceeds a certain execution time, it will block the region corresponding to the transaction thread for writing. To address this issue, we provide a long transaction region to store tuple versions generated by long transactions. If the txnId associated with a thread in the Last Commit Transaction Table is too small, it is considered a long transaction in the system. The system then copies the tuple versions associated with that transaction to the long transaction region for further processing.

In summary, the transaction processing in FIR has three major advantages compared to the previous log-free mode [13]: (1) No need to modify the last persisting record during the commit. Instead, the commit signals of multiple transactions are kept in a single NVM write. (2) Our regions are logically partitioned, rather than physically, reducing contention at specific physical locations in DRAM under high concurrency. (3) The time-ordered storage also increases the opportunities for NVM sequential writes.

4.3. Index Leaf Checkpoint with Time Cost Model

According to the conclusion in Section 2, the DRAM index slows down the system's recovery due to the rebuilding of indexes. For this issue, we propose the index leaf checkpoint based on a time cost model to rebuild indexes and the redo/undo strategy that can quickly locate the crash position using time-ordered tuple versions.

4.3.1. Incremental Index Leaf Checkpoint

FIR utilizes a two-step timestamp-based algorithm to mark and flush incremental checkpoints, as shown in Algorithm 2. We maintain the attribute *t_access* for the nodes of the index, indicating the time of the last write operation accessed, where *t_p* represents the time of the previous checkpoint and *t_c* represents the time of the current checkpoint.

In the first step (Function MarkNode), when accessing nodes with write operations, the working thread assigns the current *txnId* to *t_access* with *atomic_store_explicit*, if *t_access* is less than *t_c* (Lines 2, 3). In the second step (Function FlushChpt), when flushing incremental checkpoints, the checkpoint thread scans the index and applies a shared lock to the *t_access* in the currently traversed node *n*(Lines 10 to 12). If the *t_p* is no more than *t_access* in *n*, it continues scanning its child nodes (Line 14) until reaching a leaf node and then puts its key and value into the *checkpoint_item* (Lines 7 to 9). If *t_access* is less than *t_p* , it indicates that all child nodes of node *n* have not been written by any transaction in the time from the previous checkpoint *t_p* to the current *t_c*. So the checkpoint thread can skip node *n* and its child nodes, storing the skipped range of nodes in a special checkpoint item called *skip_item* (Lines 15, 16). Finally, after scanning the index and obtaining a key-pair sequential list, the checkpoint thread merges the data from the previous checkpoint in NVM with it, resulting in a complete checkpoint (Line 19). The advantage of this algorithm is that there is no need to traverse the entire index's leaf nodes. Instead, it determines whether to skip child nodes based on the timestamp of the parent node.

Algorithm 2: Incremental Checkpoint Functions

```
1  Function MarkNode(node, t_c)
2      if node.taccess < t_c then
3          atomic_store_explicit (&n.t_access, t_c, memory_order_relaxed);
4  Function GetIncrement(nodes[], t_p, chpt_arr)
5      for size_t i ← 0 to nodes.count - 1 do
6          n ← &nodes.nodes[i];
7          if n.type == LEAF then
8              chpt_arr.add(checkpoint_item, key, value); // Add leaf node
9              continue
10         pthread_mutex_lock(n.t_access_lock);
11         flag ← t_p ≤ n.t_access;
12         pthread_mutex_unlock(n.t_access_lock);
13         if flag then
14             GetIncrement(n.childs, t_p, chpt_arr);
               // Process child nodes
15         else
16             chpt_arr.add(skip_item, n.min, n.max);
               // Add skip item
17 Function FlushChpt(last_chpt, chpt_arr, t_p, root)
18     GetIncrement(root.childs, t_p, chpt_arr);
       // Merge and persist checkpoints
19     MergeAndPersist(last_chpt, chpt_arr);
```

4.3.2. Consistency Handling During Checkpointing

When a checkpoint is initiated, the system continues running, and new versions generated by transactions may raise concerns about the consistency of checkpoint data. We address this by discussing the following scenarios:

- **Committed Insert and Update Operations:**
  For the version data generated by committed insert and update operations, the corresponding items are already included in the index. We choose to write these items directly into the index checkpoint. This does not impact the final recovery results, as during the system's redo process, the index items corresponding to the committed tuples are exactly what we need.
- **Committed Delete Operations:** For committed delete operations, since the items in the index nodes have already been removed or marked as deleted, the system will directly skip them during the index checkpointing process. As these delete operations have already been committed, this will not affect the consistency of the final recovery.
- **Uncommitted Insert and Update Operations:** For uncommitted insert and update operations, when the checkpoint thread encounters index items generated by these operations, it accesses the corresponding version's *tuple_header* through the item. If *tuple_header.end_ts* < *MAX*, it indicates that the transaction generating the tuple version has not yet been committed. In such cases, the thread traces the version chain to find the last committed version of the tuple and writes it to the checkpoint, ensuring consistency.
- **Uncommitted Delete Operations:** For uncommitted delete operations, the corresponding index items are not deleted but are marked as pending deletion. The

checkpoint thread can still locate the corresponding tuple version through the index item and trace it back to its last committed version, ensuring consistency.

4.3.3. Time Cost Model

As mentioned earlier, FIR's index checkpoints come in two types: incremental and full index checkpoints. Figure 6 illustrates the influence of the write skew factor (ZipF) on different types of index checkpoints (incremental or full) in FIR. We observe that when ZipF exceeds a value between 0.7 and 0.8, performing incremental checkpoints becomes more cost-effective than full checkpoints. This is because, under this condition, write operations are concentrated on a very small range of index entries, leading to a smaller size of incremental checkpoints. On the contrary, when the range of index writes is very uniform, performing a full checkpoint is more cost-effective. Therefore, we hope that before each flush of the index checkpoint, the system will make a decision to achieve the optimal choice.

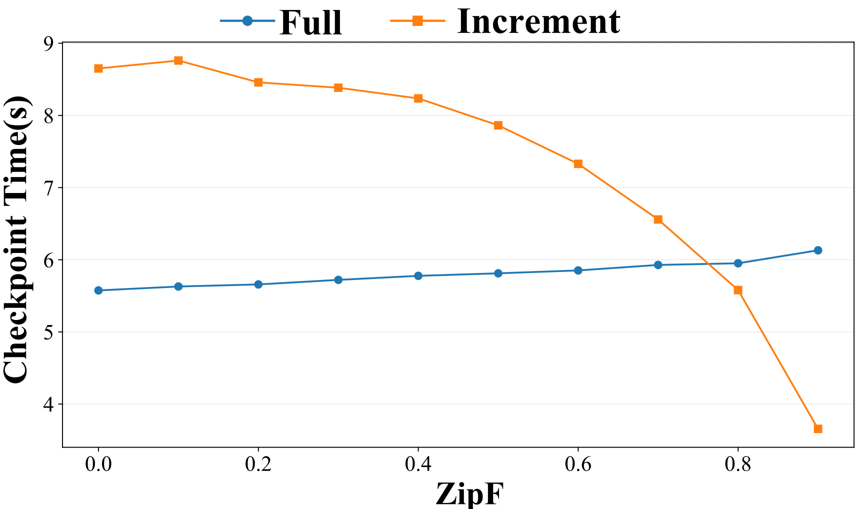


Figure 6. The impact of ZipF on checkpoint time for YCSB: 50% read, 50% update.

To improve the efficiency of index checkpoint flush and ensure that the system can choose the required checkpoint mode, we introduce a statistical time cost model. This model calculates the ZipF α of the current workload by sampling data, using them to estimate the time T_{Full} for a full checkpoint and the time $T_{\triangle}$ for an incremental checkpoint when the index checkpoint starts. If $T_{\triangle} > T_{\text{Full}}$, a full checkpoint is performed; otherwise, an incremental checkpoint is executed.

Cost of a Full Checkpoint. When flushing a full checkpoint, the system needs to traverse all leaf nodes of the index, obtain key–value pairs from the leaf nodes to construct a sequential list, and keep the list in NVM. Let N be the number of index entries, let the entire index be an M-ary tree with a height of h, let f be the time to access a node, s be the size of a key–value pair, let d be the time of a single DRAM write, and let p be the time of a single NVM write. Without considering the blocking of threads during index scanning, the time overhead T_{Full} to scan the entire index and generate a full checkpoint is in the following range:

$$T_{\text{Full}} = 0.7f(M^{(h+1)} - 1) + Ns\left(\frac{d}{64} + \frac{p}{256}\right) \quad (1)$$

where 0.7 is the average fill factor of Masstree [47], and $M^{(h+1)} - 1$ represents the number of nodes in a full tree [59]. The minimum write sizes for DRAM and NVM are 64 bytes and 256 bytes, respectively. We can use $\frac{Ns}{64}$ and $\frac{Ns}{256}$ to denote the number of times writing to DRAM and NVM [9,16,17] .

Cost of Incremental Checkpoint. We measure the write skew factor of the update workloads for each table over a period of time by calculating the Gini coefficient [18,60] through sampling statistics, denoted as α. We uniformly collect 1% of the data as samples for each table, with a quantity of n. During system runtime, at regular intervals of time t, we record the total update count C for the samples and the update count $updates_i$ for the i-th record. The skewness of updates, α, is then calculated as follows:

$$\alpha = 1 - \sum_{i=1}^{n} \left(\frac{\text{updates}_i}{C} \right)^2 \quad (2)$$

We assume that the number of index entries is N, and the entire index is an M-ary tree with a height of h. In incremental checkpoints, the traversed numbers of nodes are calculated as $0.7f(1-\alpha)(M^{h+1}-1)$.

Also, it needs to add the time it takes to merge the last checkpoint. This process involves reading data from NVM, merging them with the incremental data in DRAM (with a complexity of $O(2N)$) [61], and then writing them back to DRAM. Assuming the values of keys are uniform, the data size to be read from NVM is approximately equal to the data size of the incremental data.

Similarly, let f be the time to access a node, let s be the size of a key–value pair, let d be the time of a single DRAM write, and let p and q be the time of a single NVM write/read. The overall cost of the incremental checkpoint is approximately

$$T_{\Delta} = (1-\alpha)[0.7f(M^{h+1}-1) + Ns\left(\frac{d}{64} + \frac{2p+q}{256}\right)] + O(2N) \quad (3)$$

Here, $O(2N)$ refers to the time complexity for merging the last checkpoint. Assuming k represents the time per unit length for merging checkpoints, this term can be replaced by $2(1-\alpha)Nk$, which accounts for the merging cost of the checkpoint. By combining Equations (2) and (3), we obtain the following condition for deciding whether to perform an incremental checkpoint:

$$\alpha > \frac{Ns\left(\frac{p+q}{256}\right) + 2Nk}{0.7f(M^{h+1}-1) + Ns\left(\frac{d}{64} + \frac{2p+q}{256}\right) + 2Nk} \quad (4)$$

When α satisfies this condition, an incremental checkpoint is performed.

4.4. Recovery

After a system crash, all data in DRAM are lost, including indexes, metadata, and runtime data. In NVM, many dirty tuple versions may remain. We need to use the information recorded in NVM to restore the system to its state before the crash. The entire recovery process has the following three steps: redo/undo transactions, rebuild indexes, and load metadata. Figure 7 illustrates a recovery instance.

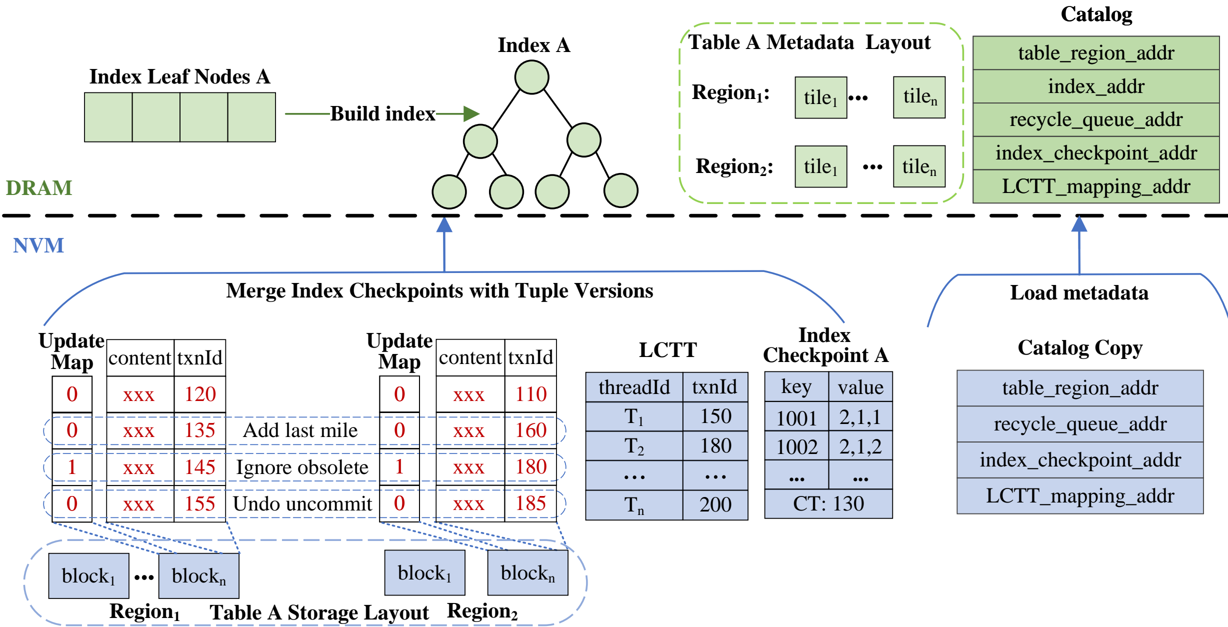


Figure 7. FIR recovery process.

4.4.1. Undo/Redo Transactions without Scanning Entire Content Data

Due to forcefully interrupting the executing transactions, these uncommitted transactions often leave numbers of dirty tuple versions on NVM. If these dirty tuple versions are not cleared, they can lead to data inconsistency. Therefore, undo operations are required to eliminate the impact of these uncommitted transactions. In FIR, leveraging the temporality of tuple versions makes it easy to perform undo operations. As mentioned in Section 4.1, all data in each region are sorted by time. This implies that all uncommitted tuple versions can only exist at the end of the region. Besides undo operations, FIR also needs to load tuple versions kept by committed transactions but not in checkpoint using redo operations. The redo/undo operations are sequential.

In Figure 7, according to *table_region_addr*, we can get the last block $block_n$ and find the first tuple version whose $txnId$ is greater than 130. Because of the time-ordered storage, from that tuple version onward, each tuple version has a $txnId$ greater than 130. For each tuple version whose $txnId$ is greater than 130 (last checkpoint time) and less than 150 (last committed transaction) in region 1 and greater than 130 and less than 180 in region 2, if the corresponding position in the update_map is 1, it indicates that the tuple version is obsolete, and the tuple version should be discarded. The remaining tuple versions, which have been committed but not written into the index checkpoint, should be inserted into the index checkpoint. For each tuple version whose $txnId$ is greater than 150 in region 1 and greater than 180 in region 2, it should be discarded directly because it has not been committed.

In traditional log-free mode, tuple versions persisted by uncommitted transactions may be located in various places of the content data. Consequently, it is necessary to traverse all tuple versions to determine the maximum committed transaction ID, identify uncommitted tuple versions, and mark them for deletion. The time complexity of the entire process is at least $O(n)$, where n is the number of tuple versions in the region. In FIR, the data that need to be scanned during recovery correspond only to the tuple versions generated from the last checkpoint to the system crash. Let n denote the total data size, and let m represent the number of tuple versions generated from the last checkpoint to the crash. Consequently, the time complexity of our recovery algorithm is $O(m)$. It is important to note that $O(n)$ is greater than $O(m)$.

Assuming the system operates for a total time T, with a data generation rate of v tuples per second, the total data size n is given by

$$n = v \times T \tag{5}$$

Let the checkpoint interval be t, and assume that a new checkpoint starts immediately after the previous one finishes. In the worst case, the number of tuple versions generated from the last checkpoint to the crash is

$$m = v \times t \tag{6}$$

As the system runs for a longer period, the ratio of checkpoint time to total running time, $\frac{t}{T}$, gradually approaches zero. Consequently, the time complexity of the algorithm approaches $O(1)$, since m becomes negligible relative to n. This demonstrates that the recovery process becomes increasingly efficient as the system runs for a longer time.

4.4.2. Rebuild Indexes with Index Leaf Checkpoint

After redo/undo transactions and loading metadata, the system cannot run directly because the indexes need to be rebuilt. In Zen's log-free mode, indexes are rebuilt by the top-down algorithm when scanning all versions. This slows down the entire recovery process. FIR can quickly build indexes using a bottom-up algorithm through the checkpoints with ordered leaf nodes. Compared to top-down index-construction algorithms, bottom-up algorithms have the following three advantages: (1) the bottom-up algorithm has a time complexity of $O(n)$, while the top-down algorithm is $O(nlog(n))$ [62]; (2) the bottom-up algorithm always builds a balanced index tree, eliminating the need for redundant operations such as splitting or merging; (3) the bottom-up algorithm is multi-thread-friendly when building indexes, with no contention about multiple threads for a single node.

In summary, FIR's entire recovery process can be completed instantly, mainly for the following reasons: (1) Throughout the undo process, FIR does not need to scan the entire table. Instead, due to the time-ordered tuple versions, the system can easily locate tuple versions that have not been committed, enabling a fast undo operation. (2) In the index rebuilding process, using index checkpoints enables multi-threaded and non-contentious construction.

4.5. Garbage Collection (GC)

As mentioned earlier, FIR sets the corresponding position in the *update_map* to 1 when updating. If all values in the *update_map* are set to 1, it indicates that all tuple versions in the block are obsolete. During the system runtime, when more than 90% of the tuple versions in a block are obsolete, the logical address of the block will be added to the *gc_array* for future garbage collection. When the length of the *gc_array* reaches a certain threshold, the garbage collection thread begins the recovery of data in the *gc_array*.

First, to avoid affecting the consistency of the index checkpoint during this process, we pause the index checkpoint work at the beginning of garbage collection. (1) For each block to be collected, the GC thread scans it and inserts the copies of non-obsolete versions into blocks of *gc_region*, modifying their index entries. Then, (2) after scanning all obsolete blocks, FIR does not immediately free the NVM space. This is because once the system crashes during the process of releasing space in NVM, the indexes rebuilt by existing index checkpoints will be unable to access the tuple versions in obsolete blocks. The right way is to perform an index checkpoint immediately at (2) and release all obsolete blocks after the checkpoint is completed.

In addition, it is a challenge to ensure that after recovery from a failure, the system continues the previous GC work without affecting correctness and without causing storage

leaks in NVM. In FIR, we divide the entire process into two stages, marked by the start and completion of the first index checkpoint after GC. If the first checkpoint is not completed, it implies that the recovered index points to unreclaimed tuple versions, requiring another round of GC after recovery. If the first checkpoint is completed, the system continues the garbage collection process from where it left off before the crash, releasing the pending reclaimed NVM storage.

In summary, our GC strategy has its advantages and irreplaceability. Firstly, traditional GC requires traversing tuple versions to find obsolete versions, which incurs significant overhead. In contrast, our strategy only scans blocks that are in the *gc_array* to be obsolete. Secondly, the space reclaimed by traditional GC is not contiguous, varies in size, and requires management, leading to contention for allocation slots. In contrast, our strategy can completely free NVM space.

5. Evaluation

In this section, we show the comparison between FIR and similar engines, covering aspects such as throughput, recovery speed, and checkpoint performance. All experiments are run in a real environment with a machine equipped with NVM.

5.1. Experimental Setup

Infrastructures. Experiments were conducted on a server equipped with an Intel Xeon Gold 5218R CPU (20 physical cores @2.10 GHz and 27.5 MB LLC) and 256 GB DDR4 DRAM, running Ubuntu 20.04.3 LTS. Each core supports two hardware threads, resulting in a total of 40 threads. There are 768 GB (128 × 6 GB) Intel Optane DC Persistent Memory NVDIMMs in the system. Optane PM has two accessibility modes, Memory mode and APP Direct mode. We choose APP Direct mode for the easy mapping of DRAM and NVM into the same virtual address space. We deploy file systems in fs-dax mode on NVM, followed by employing libpmem within PMDK to establish mappings between NVM files and a process's virtual memory. To ensure data persistence to NVM, we utilize *clwb* and *sfence*. The entire codebase is developed in C/C++ and compiled using gcc 7.5.0.

Control Groups. The control groups for our experiment include DRAM-DB (no log), DRAM-DB (log), WBL (NVM Idx), Zen, WBL (DRAM Idx), FIR, FIR (Full checkpoint), and IACoW.

To eliminate interference from modules such as transaction management and query processing and achieve the goal of controlling variables, we choose DB×1000 [63] as the runtime platform for the mentioned NVM engines. DB×1000 is a DRAM database prototype that uses a row-oriented format and includes methods of concurrency control. We implemented the above engines in this platform, and we control their use of Masstree for index and MVTO for concurrency control. Also, we import libpmem within PMDK to support NVM read and write.

DRAM-DB (No Log): The original version of DB×1000 without logs and checkpoints, lacking recovery support, is used as a baseline.

DRAM-DB (Log): A version of DB×1000 that implements logs and checkpoints supports recovery with persistent data stored in NVM.

WBL (NVM Idx): We implemented WBL in DB×1000 following the design in [12], where the index is stored in NVM. WBL uses out-of-line transaction states for recovery.

WBL (DRAM Idx): This is the implementation of WBL with the index stored in DRAM.

Zen: We implemented Zen, the state-of-the-art NVM OLTP Engine, in DB×1000 following the design in [13]. Zen stores the index in DRAM and uses log-free mode for recovery.

FIR: The engine is described in this paper.

FIR (Full Checkpoint): FIR with full index checkpoints.

FIR (IACoW): Building upon FIR, we replace the original index checkpoints with IACoW used for comparing the differences between different index checkpoints.

To eliminate interference, we disable the DRAM caches of tuple versions in the above engines. Additionally, there are many other NVM OLTP Engines, such as FOEDUS, Hyrise, Falcon, etc. The experimental results in [28] reveal that FOEDUS has a much lower throughput and recovery performance compared to Zen. Hyrise is similar to WBL and exhibits poor throughput [29], while Falcon leverages new eADR-enabled NVM hardware [3], with its research direction belonging to a different dimension from FIR.

Workloads. We utilize two benchmarks, YCSB [64] and TPCC [65], to evaluate our system.

YCSB (Yahoo! Cloud Serving Benchmark) is a commonly used key–value benchmark tool designed to assess the performance of the database and storage systems under read and write workloads. It allows for a variety of workload characteristics through configurable parameters. These parameters include the number of requests in each transaction, the proportion of read, write, and read–modify–write (RMW) requests, and the skew factor that determines the distribution of requested keys in the ZipF distribution. In our experiments, the YCSB contains a single table and the size of every tuple version is 100 bytes with an 8-byte primary key. Every transaction consists of 16 requests.

TPCC (Transaction Processing Performance Council Benchmark C) is a benchmark for transaction processing performance. It is a database performance testing standard defined and maintained by the Transaction Processing Performance Council (TPC). TPCC is primarily used to simulate and evaluate the performance of database systems under high transaction loads with multiple users. It includes nine tables designed to emulate an online order processing application and involves five mixed transactions: Neworder (45%), Payment (43%), Ordstat (4%), Delivery (4%), and StockLevel (4%).

5.2. Transaction Performance

5.2.1. TPCC Experiments

Throughput. To evaluate the throughput of FIR and other engines under high concurrency, we conduct tests using the TPCC workload. We set the TPCC data size to 500 warehouses with 30 threads. The experimental results are shown in Figure 8a, where the x-axis represents different engines and the y-axis represents the throughput ratio of each engine to the baseline (where we consider the DRAM-DB (No Log) as the baseline). We observe that FIR achieves approximately 85% of the baseline, while the complete engine DRAM-DB (Log) only reaches around 65% of the baseline. It is worth noting that FIR is close to Zen in performance but slightly lower. Although FIR incurs additional overhead from flushing checkpoints, this process does not block index read and write operations, resulting in minimal system overhead. Furthermore, the worst-performing system is WBL with the index placed in NVM, achieving only 49% throughput. This is because the index needs to be frequently read and written, and a large number of small writes as well as random write operations are not friendly to NVM. FIR (Full Checkpoint) and FIR (IACoW) both exhibit lower throughput compared to FIR, at 76.5% and 68.8%, respectively.

Scalability. We also tested the scalability of various engines under the TPCC workload. The experimental results are shown in Figure 8b, where the x-axis represents the number of threads ranging from 1 to 60, and the y-axis represents the throughput of TPCC under each thread, measured in millions per second. We observe that FIR and Zen exhibit the best performance among all engines, except for DRAM-DB (No Log). At a thread count of 40, all engines reach a bottleneck, with FIR achieving a throughput of 1.61 million/s.

Latency. As shown in Figure 8c, DRAM-DB (No Log) exhibits the best performance in system latency. FIR outperforms Zen in average latency but is slightly outperformed by Zen in 99th-percentile latency, with a difference of less than 1%. Notably, FIR (IACoW)

has significantly higher latency compared to FIR, reaching its peak in the 99th-percentile payment transactions at 2.2× that of FIR. Since IACoW persists throughout the entire index, and it significantly reduces the efficiency of checkpoint flushing, leading to a substantial increase in 99th-percentile latency. WBL (NVM) exhibits poor latency performance, with 99th-percentile latency ranging from 1.5× to 2.6× that of FIR. Meanwhile, DRAM (Log) performs the worst in the 99th-percentile latency, ranging from 1.72× to 3.5× that of FIR.

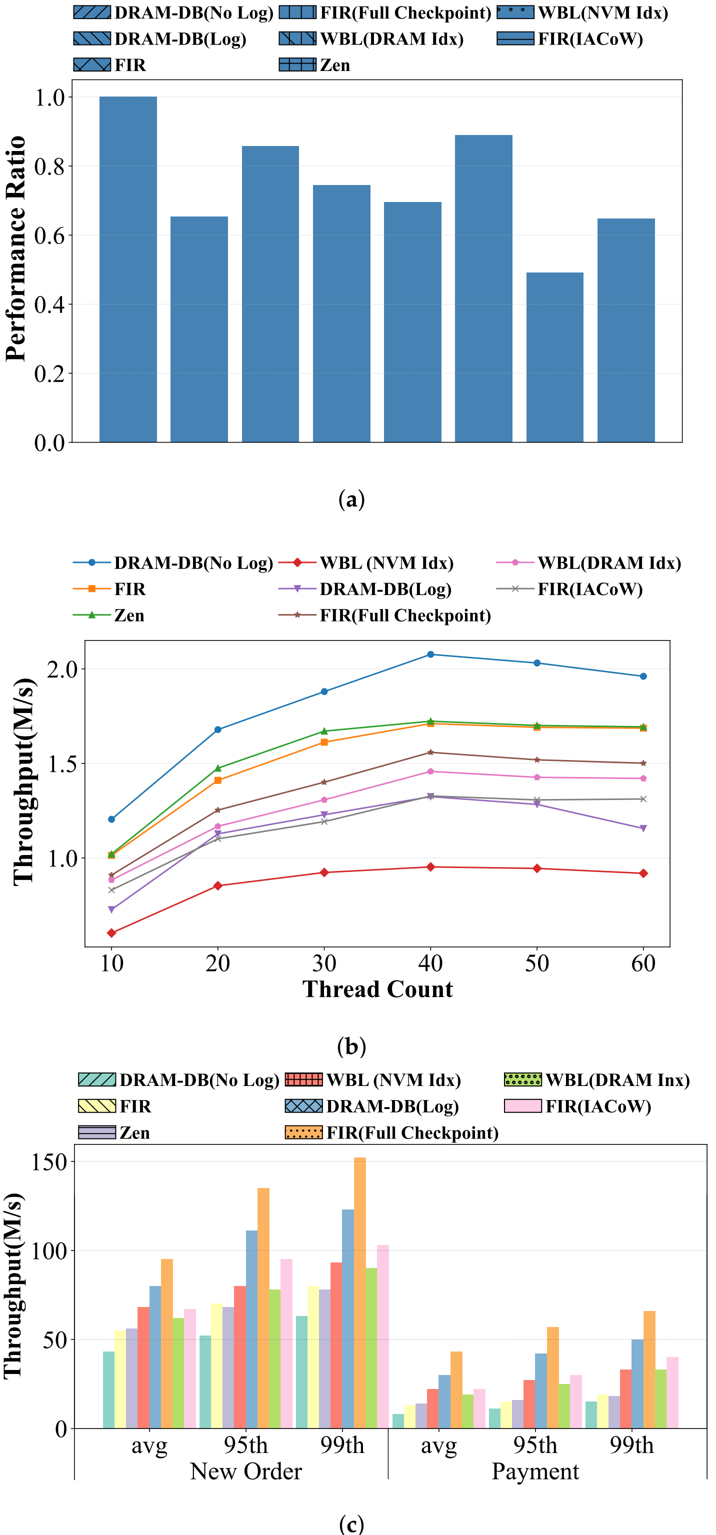


Figure 8. TPCC Performance with different OLTP Engines. (**a**) Performance ratio on TPCC; (**b**) scalability on TPCC; (**c**) latency on TPCC.

5.2.2. YCSB Experiments

Standard YCSB workload. We control the number of threads to be 30 and run the standard YCSB workloads, including (A) 50% read, 50% update; (B) 95% read, 5% update; (C) 100% read; (D) 95% read recent, 5% insert; and (E) 95% scan, 5% insert. The dataset contains a single table with 100 million tuples. Additionally, we set three ZipF parameters (α = 0, 0.5, 0.9), representing completely uniform access, moderately uniform access, and a small number of tuples accessed, respectively.

Figure 9a–c depict the experimental results for ZipF factors of 0, 0.5, and 0.9, respectively. The x-axis represents different workloads, while the y-axis represents transaction throughput in transactions per second. Regardless of running different types of transactions or different ZipF factors, WBL (NVM Idx) always exhibits the poorest performance among the tested engines. It not only frequently writes logs to NVM and modifies metadata in tuple versions but also maintains the index in NVM, significantly slowing down the system's performance. In contrast, WBL (DRAM Idx) significantly improves throughput by maintaining the index in DRAM, achieving approximately 1.4×–1.5× the throughput of WBL (NVM Idx).

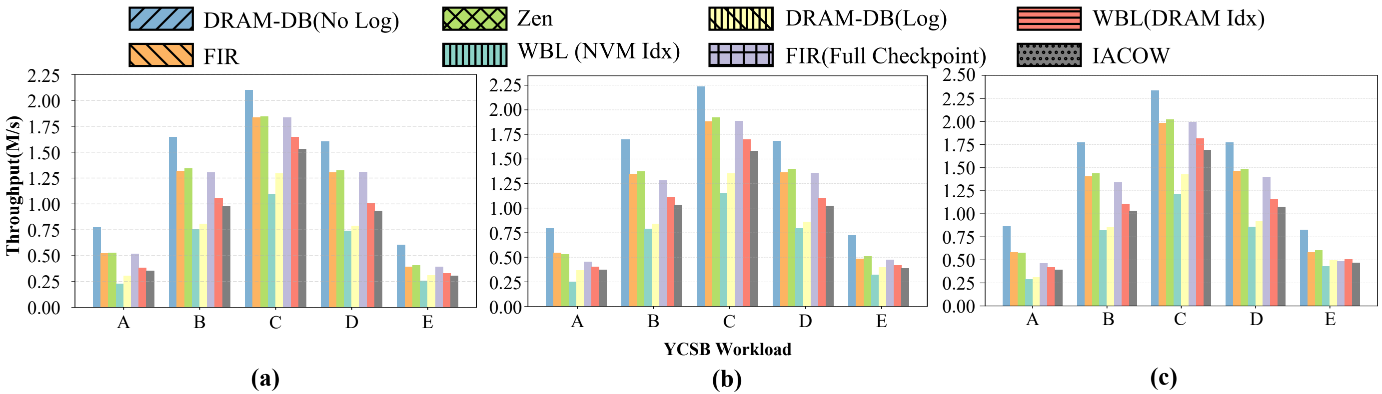


Figure 9. TPCC performance with different OLTP Engines. (**a**) ZipF = 0; (**b**) ZipF = 0.5; (**c**) ZipF = 0.9.

DRAM-DB (No Log) does not involve read or write operations on NVM; we simply use it as a baseline for comparison with other engines, and it exhibits the best experimental results. However, when it acquires recovery capabilities and becomes a usable engine (DRAM-DB (Log)), its performance drops significantly. As the main design of the DRAM-based OLTP Engine, the performance of DRAM-DB (Log) is even lower than that of WBL (DRAM Idx) and only slightly higher than WBL (NVM Idx). This is because DRAM-DB (Log) needs to flush data to NVM twice, once for the checkpoint and once for the redo log.

As a DRAM-DB index checkpoint scheme, we implemented IACoW in NVM-DB called FIR (IACoW) to demonstrate that this approach is not suitable for NVM-DB. The experimental results confirm our perspective, as shown in the figure. The throughput of FIR (IACoW) is only 67% of the FIR under write-intensive workloads and slightly better under read-intensive workloads, reaching 85% of FIR. This is because IACoW's index is multi-versioned, and during checkpoint flushing, the entire index structure is serialized and then persisted to NVM, slowing down the entire system. In contrast, FIR keeps only a part of the index (incremental leaf nodes) as a checkpoint, significantly reducing the pressure on NVM writes.

In the above experiments, FIR and Zen are nearly comparable, achieving around 67–85% of the baseline. In most cases, FIR's performance is slightly lower than Zen's. However, when ZipF is 0.9, running workload A, FIR outperforms Zen. This is because, on the one hand, the larger the ZipF, the fewer nodes need to be flushed for incremental

checkpoints in FIR, reducing the impact of checkpoints on runtime performance. On the other hand, Zen maintains more metadata in line with the tuple version than FIR. The more updates are performed, the more frequently the metadata will be updated. Therefore, workloads with a significant number of updates, like workload A, are less friendly to Zen. As for FIR with full checkpoint implementation, with ZipF increasing, its performance is gradually lower than that of FIR, especially under write-heavy workloads. When executing workload A with ZipF at 0.9, its throughput is only 68% of FIR's.

5.2.3. Comparison to the NVM Index

To compare the performance of the index in FIR with NVM indexes, we implement some of the most advanced NVM indexes in DB×1000, including Hyrise, Roart, Nbtree, and wB+-Trees. Therefore, our experimental platform is DB×1000, and the experimental subjects include Masstree (Baseline, DRAM index with no index checkpointing), FIR, and the aforementioned NVM indexes. We use 20 threads to perform 1 billion random insert operations and record throughput per second. The initial index size is 100 million entries, and the key size is 64 bits. In FIR, we open index checkpointing to allow index persistence. The entire experiment results are shown in Figure 10. As we can see, the throughput of FIR is slightly lower than the Baseline, at 90.04% of its performance. However, compared to the other NVM indexes, FIR shows a significant performance advantage, being approximately 1.14× to 1.78× higher than the other indexes.

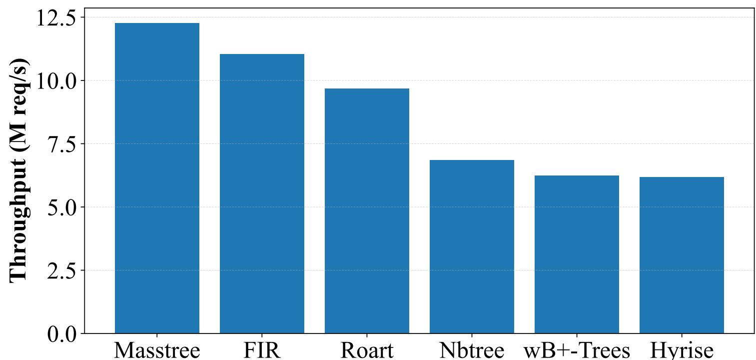


Figure 10. Performance of FIR index and advanced NVM indexes.

In conclusion, FIR demonstrates excellent throughput. In most scenarios of TPCC and YCSB, the performance of FIR is very close to Zen, and in some cases, it even outperforms Zen. Compared to other similar engines, FIR has a clear advantage. The index performance of FIR is also significantly higher than advanced NVM indexes due to its use of a DRAM index.

5.3. Recovery Performance

We evaluate the recovery performance of the above OLTP Engines using the TPCC benchmark. Firstly, we load the same number of warehouses for each OLTP Engine and then run the TPCC workloads for the same period of time. Due to differences in transaction execution performance between these engines, we control the TPCC throughput of all engines to be the same low value to ensure fairness. This ensures that after running for the same amount of time, engines can achieve the same data size. We set two initial data sizes in our experiments, 128 warehouses (12 GB) and 2048 warehouses (190 GB). The TPCC transaction throughput is set to 300,000/s, resulting in data sizes of 83 GB and 261 G after running for 100 s. However, reducing throughput does not imply that we will adjust the frequency of other modules, such as the time interval of checkpoint flush in FIR or the garbage collection frequency in WBL. These parameters directly impact recovery perfor-

mance, and we maintain consistency with the previous experiment settings. Subsequently, we evaluate recovery time in two scenarios: One is Unplanned Crash, where the engine is terminated suddenly according to a pre-set time. The other is Manual Shutdown, where the engine is not terminated after workloads are finished. It terminates the engine after a sufficient period, allowing all transactions to be committed and checkpoints (if exist) to be the latest versions. Finally, we restart the engine and record the engine recovery time. To comprehensively evaluate the recovery performance of each engine, we record the time for rebuilding indexes, the time for redo/undo operations, and the time for loading checkpoints and metadata.

Unplanned Crash. The experimental results are shown in Figure 11, where (a) and (b) represent the results for initial data sizes of 128 warehouses and 2048 warehouses, respectively. We observe that under the Unplanned Crash scenario, WBL, FIR (IACoW), and FIR exhibit a significant recovery speed advantage, being 43.6×–54.5× compared to other engines. Combining these results with previous experiments, while Zen has a slightly higher throughput than FIR, its recovery speed is only 1.8–5.4% of FIR. This is because FIR can quickly rebuild the index through index checkpoints, requiring only 1.45 s even when the data size is 261 GB. In contrast, Zen needs to scan all data to build the index and perform undo operations, and its recovery speed is only slightly better than that of DRAM-DB (Log), which involves loading checkpoints, replaying redo logs, and rebuilding the index. When an initial warehouse is 128, the recovery speed of FIR even exceeds that of WBL (NVM Idx). This is because, with a smaller recovery data size, FIR spends less time building the index, while WBL (NVM Idx) needs to perform undo operations based on logs. In the crash mode, the engine leaves an amount of logs, resulting in a longer recovery time. At an initial warehouse count of 2048, although the recovery speed of FIR is slightly lower than that of WBL, it is much higher than that of other engines. Interestingly, FIR's recovery speed at 2048 warehouses is 2× that of FIR (IACoW). This is because, as the size of data increases, the time for FIR (IACoW) to refresh an index checkpoint is much higher than FIR. Therefore, when the system crashes, FIR (IACoW) needs more time to replay the redo log, as shown in the figure.

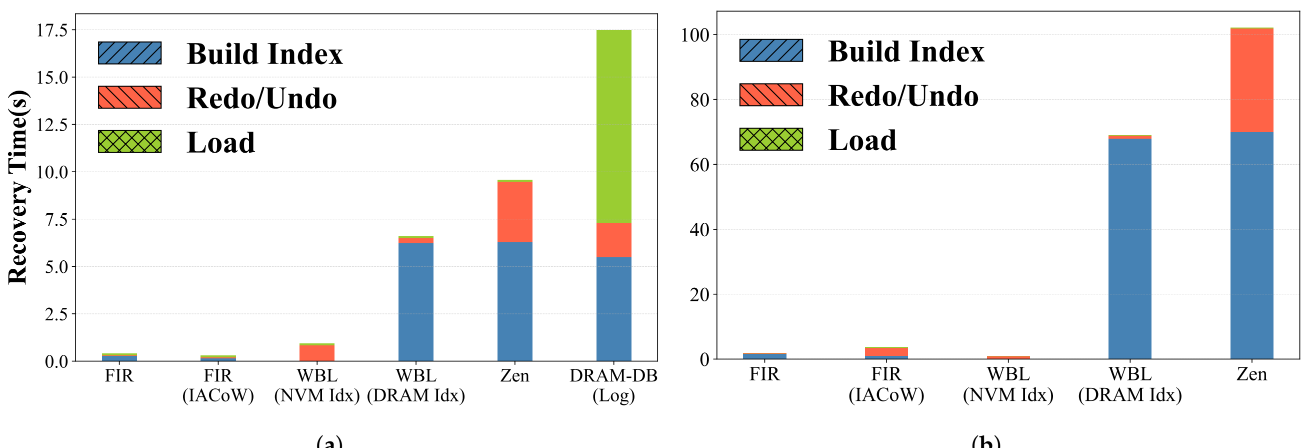


Figure 11. Recovery with Unplanned Crash. (**a**) Warehouse = 128. (**b**) Warehouse = 2048.

Manual Shutdown. The experimental results are shown in Figure 12, where (a) and (b) represent the results for initial data sizes of 128 and 2048 warehouses, respectively. We observe that under the Manual Shutdown scenario, the redo/undo time for all systems is 0. This is because, after a long period of time, all engines reach a state where all transactions are committed. In the specific performance, WBL (NVM Idx) exhibits a significantly faster recovery speed than other systems. In both initial conditions of 128 warehouses and 2048 warehouses, the recovery time is less than 0.1 s. This is because WBL (NVM Idx) does not

need to recover the index (as its index is on NVM) and requires no undo. It only needs to load some metadata into DRAM to complete recovery. The next is FIR (IACoW). Although the recovery speed of FIR is slower than WBL (NVM Idx) and FIR (IACoW) in the Manual Shutdown scenario, the advantage is still very evident compared to other engines. The recovery speed of FIR is 15.5×–43.1× faster than other engines (excluding WBL (NVM Idx) and FIR (IACoW)). Zen, due to the need to scan all tuple versions to build indexes, is time-consuming, despite not needing to undo operations in this scenario. Its recovery speed is only 5.3–6.4% of FIR. It is worth noting that although WBL (NVM Idx) is much faster than FIR and FIR (IACoW), considering the entire startup time of DB×1000 (which takes about 2 s), the entire recovery time of the three engines is in the same order of magnitude. Also, we can see that the speed of building the index for FIR and FIR (IACoW) is actually quite close. This indicates that the speed of constructing an index in DRAM using sorted leaf nodes from NVM is not significantly slower than loading the entire index from NVM and then deserializing it in DRAM.

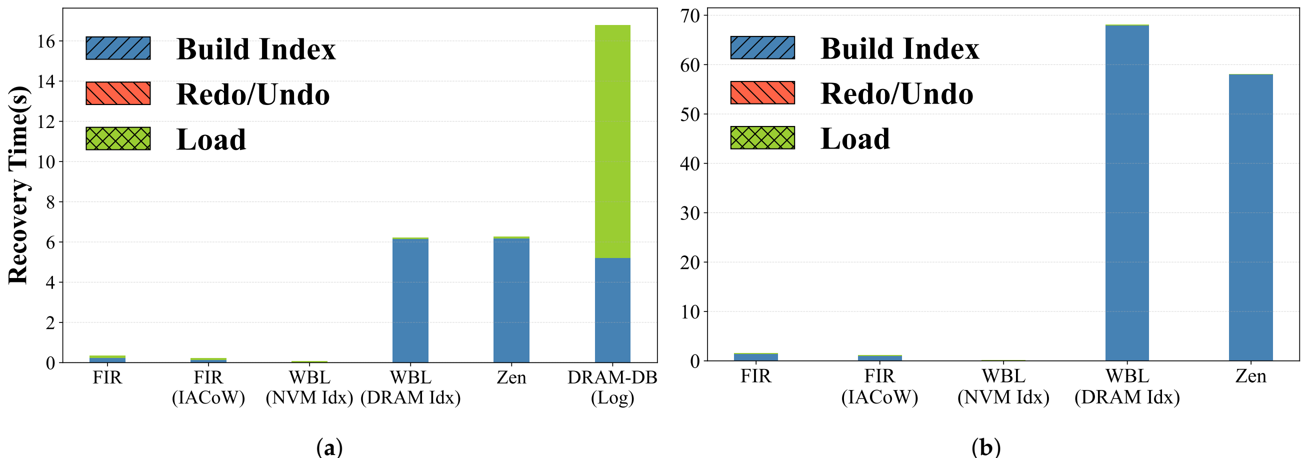


Figure 12. Recovery with Manual Shutdown. (**a**) Warehouse = 128. (**b**) Warehouse = 2048.

We can draw the following conclusions. First, FIR is able to maintain a high runtime throughput competitive with the best-performing NVM OLTP Engine (Zen) while achieving significantly better recovery performance than all other engines except WBL (NVM Idx). Interestingly, when we attempted to move the index in WBL (NVM Idx) to DRAM, trying to improve its throughput, its recovery speed dropped by several orders of magnitude. This is because even though WBL can perform undo operations quickly, it is impossible for the system to run immediately without complete indexes. Although Zen outperforms FIR in terms of throughput, its recovery speed is far inferior to FIR. Finally, as an index checkpoint solution in DRAM, while IACoW has a recovery speed close to that of FIR, its throughput is much slower than FIR's. Therefore, it is not suitable for NVM OLTP Engines.

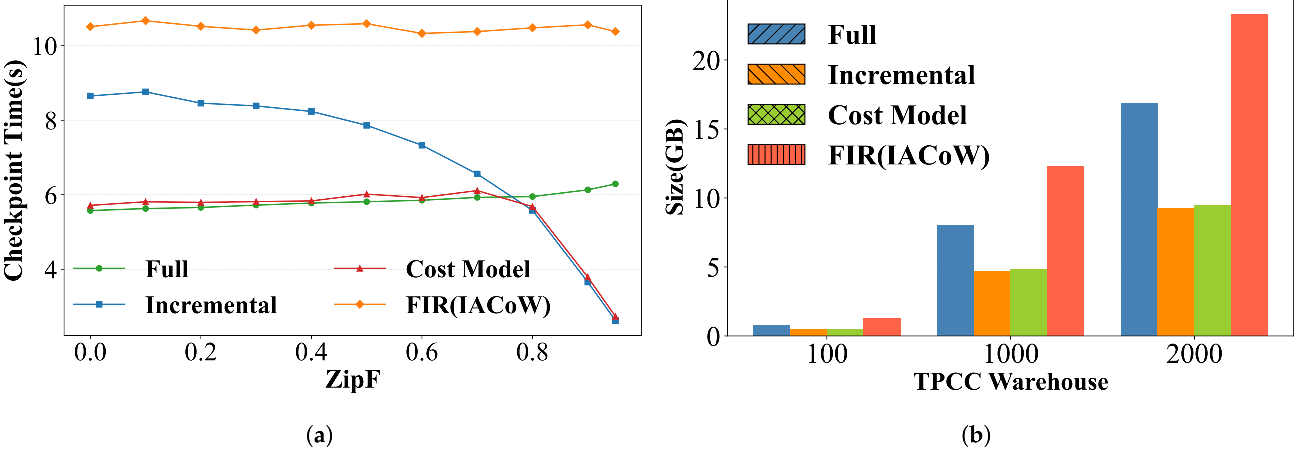
5.4. Index Checkpoint Performance

There are five control groups used in the following experiments. The first is Full Checkpoint (Full), the FIR supporting only the Full Checkpoint . The second is Incremental Checkpoint (Incremental), the FIR supporting only the Incremental Checkpoint. The third is the Cost Model, the FIR version using the time cost model to choose full or incremental checkpoints. The fourth is FIR (IACoW). The last is Baseline, the FIR without a checkpoint.

5.4.1. Checkpoint Speed

Figure 13a shows the variation in checkpoint speed with increasing YCSB ZipF. We use YCSB workload (A), with a total of 100 million tuples. The x-axis represents the increasing ZipF, and the y-axis represents the flush time of the checkpoint. We evaluate the full,

incremental, and cost models in this experiment. We observe that whether the range of requested keys is uniform or concentrated, FIR with a time cost model can always choose a better way of checkpoint, maintaining a relatively fast speed, while full and incremental models cannot always maintain the best performance. Although the speed of FIR (IACoW) remains stable with ZipF variations, it is only at 36–54% of the cost model.



(**a**) (**b**)

Figure 13. Checkpoint speed and scale under different models. (**a**) Checkpoint speed. (**b**) Checkpoint scale.

5.4.2. Checkpoint Scale

Figure 13b shows the variation in checkpoint size with different TPCC warehouse counts. We also evaluate the full, incremental, and cost models in this experiment. We observe that as the warehouse count increases, the space utilization of the incremental model improves. At 2000 warehouses, the size of the incremental model is only 60% of the full model. After the selection of the cost model, the size of the index checkpoint is 91% to 95% of that of the incremental odel. Due to IACoW keeping the entire index tree as a checkpoint to NVM, its space overhead is much larger than the other modes.

5.4.3. The Impact of Checkpoints on Runtime Performance

Figure 14 illustrates the impact of checkpoints on throughput when FIR runs TPCC (initially 100 warehouses, 40 threads) in different checkpoint modes, baseline, full, cost model, and FIR (IACoW). The x-axis represents the runtime, and the y-axis represents throughput. We observe that the cost model has a smaller impact on throughput compared to the full model. Its average performance can reach around 90% of the baseline.

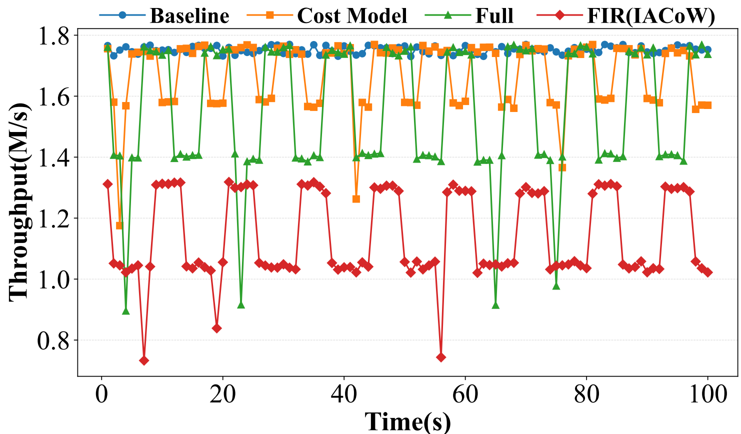


Figure 14. Throughput over 100 S.

6. Discussion

Applicability of FIR. Although various types of NVM hardware are available, such as Intel Optane Persistent Memory and emerging technologies like Phase Change Memory (PCM) [19], Magnetoresistive RAM (MRAM) [66,67], and Resistive RAM (ReRAM) [68], they still exhibit a performance gap compared to DRAM in terms of access latency and read/write speed, particularly in random write operations. In NVM OLTP Engines, indexes experience a high volume of random read and write operations under concurrent workloads. Storing indexes in DRAM can significantly enhance system performance, but it requires considerable time to rebuild the indexes during failure recovery. Conversely, storing indexes in NVM leverages its persistence to eliminate the need for index reconstruction during recovery. However, the current performance of NVM-based indexes is suboptimal, falling far short of the performance of DRAM-based indexes. As an NVM OLTP Engine, FIR ensures high system throughput while enabling rapid failure recovery. Our solution effectively compensates for the write performance shortcomings of NVM compared to DRAM. In future work, we will continuously integrate the latest NVM index implementations into our system and compare them with FIR.

A More Efficient GC Mechanism. In FIR, garbage collection (GC) is an essential component that ensures the sequential organization of tuple versions in storage by reclaiming data in block units. However, the frequent reclamation of obsolete versions causes the GC threads to continuously consume system resources, slowing down the system and negatively impacting performance. Simply reducing the GC frequency directly would not only lead to inefficient reclamation and higher NVM utilization but also result in longer version chains, ultimately affecting normal data access. As part of future work, we aim to design a more efficient GC mechanism that ensures reclamation efficiency while minimizing its impact on system performance.

7. Conclusions

OLTP Engines for non-volatile memory (NVM) can achieve either high throughput or instant recovery. However, existing designs struggle to achieve both. Indexes in DRAM significantly keep system throughput, but they require a large amount of time to rebuild during recovery. FIR uses index checkpoints based on a cost model to obtain fast index rebuilding, enabling instant recovery. Meanwhile, it keeps indexes and transaction metadata in DRAM, while NVM only stores the tuple heap and a small amount of append-only metadata for undo, ensuring high throughput.

Author Contributions: Conceptualization, J.W. and X.G.; Methodology, J.W., Y.X. and X.G.; Software, J.W.; Validation, J.W., Q.Z. and Y.X.; Formal analysis, J.W.; Investigation, J.W.; Resources, J.W.; Data curation, J.W.; Writing—original draft, J.W.; Writing—review & editing, Q.Z., Y.X. and X.G.; Visualization, Q.Z.; Supervision, X.G.; Funding acquisition, X.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by National Natural Science Foundation of China (61572194, 61672233).

Data Availability Statement: The original data presented in this study are openly available in GitHub at [https://github.com/w1397800/FIR].

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Chen, J.; Jindel, S.; Walzer, R.; Sen, R.; Jimsheleishvilli, N.; Andrews, M. The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database. *Proc. VLDB Endow.* **2016**, *9*, 1401–1412. [CrossRef]
2. Diaconu, C.; Freedman, C.; Ismert, E.; Larson, P.A.; Mittal, P.; Stonecipher, R.; Zwilling, M. Hekaton: SQL server's memory-optimized OLTP Engine. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 22–27 June 2013; pp. 1243–1254.
3. Stonebraker, M.; Weisberg, A. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* **2013**, *36*, 21–27.
4. Funke, F.; Kemper, A.; Mühlbauer, T.; Neumann, T.; Leis, V. Hyper beyond software: Exploiting modern hardware for main-memory database systems. *Datenbank-Spektrum* **2014**, *14*, 173–181. [CrossRef]
5. Lahiri, T.; Neimat, M.A.; Folkman, S. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Eng. Bull.* **2013**, *36*, 6–13.
6. Faerber, F.; Kemper, A.; Larson, P.A.; Levandoski, J.; Neumann, T.; Pavlo, A. Main memory database systems. *Found. Trends Databases* **2017**, *8*, 1–130. [CrossRef]
7. Larson, P.A.; Levandoski, J. Modern main-memory database systems. *Proc. VLDB Endow.* **2016**, *9*, 1609–1610. [CrossRef]
8. Intel Optane DC Persistent Memory Architecture and Technology. Available online: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html (accessed on 23 January 2023).
9. Akram, S. Exploiting Intel Optane Persistent Memory for Full Text Search. In Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management, Virtual, 22 June 2021; pp. 80–93.
10. Huang, J.; Schwan, K.; Qureshi, M.K. NVRAM-aware logging in transaction systems. *Proc. VLDB Endow.* **2014**, *8*, 389–400. [CrossRef]
11. Kimura, H. FOEDUS: OLTP Engine for a thousand cores and NVRAM. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, VIC, Australia, 31 May–4 June 2015; pp. 691–706.
12. Arulraj, J.; Perron, M.; Pavlo, A. Write-behind logging. *Proc. VLDB Endow.* **2016**, *10*, 337–348. [CrossRef]
13. Liu, G.; Chen, L.; Chen, S. Zen: A high-throughput log-free OLTP Engine for non-volatile main memory. *Proc. VLDB Endow.* **2021**, *14*, 835–848. [CrossRef]
14. Schwalb, D.; BK, G.K.; Dreseler, M.; Faust, M.; Hohl, A.; Berning, T.; Makkar, G.; Plattner, H.; Deshmukh, P. Hyrise-NV: Instant recovery for in-memory databases using non-volatile memory. In Proceedings of the Database Systems for Advanced Applications: 21st International Conference, DASFAA 2016, Dallas, TX, USA, 16–19 April 2016; Proceedings, Part II; Springer International Publishing: Berlin/Heidelberg, Germany, 2016; pp. 267–282.
15. Ji, Z.; Chen, K.; Wang, L.; Zhang, M.; Wu, Y. Falcon: Fast OLTP Engine for Persistent Cache and Non-Volatile Memory. In Proceedings of the 29th Symposium on Operating Systems Principles, Koblenz, Germany, 23–26 October 2023; pp. 531–544.
16. Condit, J.; Nightingale, E.B.; Frost, C.; Ipek, E.; Lee, B.; Burger, D.; Coetzee, D. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*; ACM: New York, NY, USA, 2009; pp. 133–146.
17. Jacob, B.; Wang, D.; Ng, S. *Memory Systems: Cache, DRAM, Disk*; Morgan Kaufmann: Burlington, MA, USA, 2010.
18. Atkinson, A.B. *Measuring Inequality*; Oxford University Press: Oxford, UK, 2015.
19. Le Gallo, M.; Sebastian, A. An overview of phase-change memory device physics. *J. Phys. Appl. Phys.* **2020**, *53*, 213002. [CrossRef]
20. Hu, D.; Chen, Z.; Wu, J.; Sun, J.; Chen, H. Persistent memory hash indexes: An experimental evaluation. *Proc. VLDB Endow.* **2021**, *14*, 785–798. [CrossRef]
21. Zhang, B.; Zheng, S.; Qi, Z.; Huang, L. NBTree: A Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems. *Proc. VLDB Endow.* **2022**, *15*, 1187–1200. [CrossRef]
22. Lersch, L.; Hao, X.; Oukid, I.; Wang, T.; Willhalm, T. Evaluating persistent memory range indexes. *Proc. VLDB Endow.* **2019**, *13*, 574–587. [CrossRef]
23. Ma, S.; Chen, K.; Chen, S.; Liu, M.; Zhu, J.; Kang, H.; Wu, Y. ROART: Range-query optimized persistent ART. In Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST 21), Virtual Event, 23–25 February 2021; pp. 1–16.
24. Chen, S.; Jin, Q. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.* **2015**, *8*, 786–797. [CrossRef]
25. Hassan, A.; Vandierendonck, H.; Nikolopoulos, D.S. Energy-efficient in-memory data stores on hybrid memory hierarchies. In Proceedings of the 11th International Workshop on Data Management on New Hardware, Melbourne, VIC, Australia, 31 May–4 June 2015; pp. 1–8.
26. Kim, H.; Seshadri, S.; Dickey, C.L.; Chiu, L. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. *ACM Trans. Storage (TOS)* **2014**, *10*, 1–21. [CrossRef]
27. Meza, J.; Luo, Y.; Khan, S.; Zhao, J.; Xie, Y.; Mutlu, O. A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory. 2013.
28. Izraelevitz, J.; Kelly, T.; Kolli, A. Failure-atomic persistent memory updates via JUSTDO logging. *ACM Sigarch Comput. Archit. News* **2016**, *44*, 427–442. [CrossRef]

29. Arulraj, J.; Pavlo, A.; Dulloor, S.R. 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, Melbourne, VIC, Australia, 31 May–4 June 2015; pp. 707–722.
30. Agrawal, R.; Jagadish, H.V. Recovery algorithms for database machines with non-volatile main memory. In Proceedings of the International Workshop on Database Machines, Berlin/Heidelberg, Germany, 19–24 June 1989; pp. 269–285.
31. Pelley, S.; Wenisch, T.F.; Gold, B.T.; Bridge, B. Storage management in the NVRAM era. *Proc. VLDB Endow.* **2013**, *7*, 121–132. [CrossRef]
32. Gao, S.; Xu, J.; He, B.; Choi, B.; Hu, H. PCMLogging: Reducing transaction logging overhead with PCM. In Proceedings of the 20th ACM international Conference on Information and Knowledge Management, Glasgow Scotland, UK, 24–28 October 2011; pp. 2401–2404.
33. Oukid, I.; Booss, D.; Lehner, W.; Bumbulis, P.; Willhalm, T. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. In Proceedings of the Tenth International Workshop on Data Management on New Hardware, Snowbird, UT, USA, 23 June 2014; pp. 1–7.
34. Wang, T.; Johnson, R. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.* **2014**, *7*, 865–876. [CrossRef]
35. Mohan, C.; Haderle, D.; Lindsay, B.; Pirahesh, H.; Schwarz, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. (TODS)* **1992**, *17*, 94–162. [CrossRef]
36. Haerder, T.; Reuter, A. Principles of transaction-oriented database recovery. *ACM COmputing Surv. (CSUR)* **1983**, *15*, 287–317. [CrossRef]
37. Malviya, N.; Weisberg, A.; Madden, S.; Stonebraker, M. Rethinking main memory OLTP recovery. In Proceedings of the 2014 IEEE 30th International Conference on Data Engineering, Chicago, IL, USA, 31 March– 4 April 2014; pp. 604–615.
38. Mohan, C.; Levine, F. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. *ACM Sigmod Rec.* **1992**, *21*, 371–380. [CrossRef]
39. Graefe, G.; Kuno, H. Definition, detection, and recovery of single-page failures, a fourth class of database failures. *arXiv* **2012**, arXiv:1203.6404. [CrossRef]
40. Sauer, C.; Graefe, G.; Härder, T. Single-pass restore after a media failure. In Proceedings of the Datenbanksysteme für Business, Technologie und Web (BTW 2015), Hamburg, Germany, 2–3 March 2015.
41. Graefe, G.; Guy, W.; Sauer, C. *Instant Recovery with Write-Ahead Logging*; Springer Nature: Berlin/Heidelberg, Germany, 2022.
42. Sauer, C. *Modern Techniques for Transaction-Oriented Database Recovery*; Gesellschaft für Informatik: Bonn, Germany, 2019.
43. Sauer, C.; Graefe, G.; Härder, T. Instant restore after a media failure. In Proceedings of the Advances in Databases and Information Systems: 21st European Conference, ADBIS 2017, Nicosia, Cyprus, 24–27 September 2017; Proceedings 21; Springer International Publishing: Berlin/Heidelberg, Germany, 2017; pp. 311–325.
44. Gong, C.; Tian, C.; Wang, Z.; Wang, S.; Wang, X.; Fu, Q.; Qin, W.; Qian, L.; Chen, R.; Qi, J.; et al. Tair-PMem: A fully durable non-volatile memory database. *Proc. VLDB Endow.* **2022**, *15*, 3346–3358. [CrossRef]
45. Zhang, M.; Hua, Y. Silo: Speculative Hardware Logging for Atomic Durability in Persistent Memory. In Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Montreal, QC, Canada, 25 February–1 March 2023; pp. 651–663.
46. Lee, L.; Xie, S.; Ma, Y.; Chen, S. Index checkpoints for instant recovery in in-memory database systems. *Proc. VLDB Endow.* **2022**, *15*, 1671–1683. [CrossRef]
47. Mao, Y.; Kohler, E.; Morris, R.T. Cache craftiness for fast multicore key–value storage. In Proceedings of the 7th ACM European conference on Computer Systems, Bern, Switzerland, 10–13 April 2012; pp. 183–196.
48. Bohannon, P.; Mcllroy, P.; Rastogi, R. Main-memory index structures with fixed-size partial keys. In Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA, 21–24 May 2001; pp. 163–174.
49. Levandoski, J.J.; Lomet, D.B.; Sengupta, S. The Bw-Tree: A B-tree for new hardware platforms. In Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE), Brisbane, Australia, 8–11 April 2013; pp. 302–313.
50. Leis, V.; Kemper, A.; Neumann, T. The adaptive radix tree: ARTful indexing for main-memory databases. In Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE), Brisbane, Australia, 8–11 April 2013; pp. 38–49.
51. Rao, J.; Ross, K.A. Making B+-trees cache conscious in main memory. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, USA, 16–18 May 2000; pp. 475–486.
52. Dean, J.; Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
53. Corporation, I. eADR: New Opportunities for Persistent Memory Applications. 2021. Available online: https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistentmemory-applications.html (accessed on 1 November 2023).
54. Zheng, W.; Tu, S.; Kohler, E.; Liskov, B. Fast databases with fast durability and recovery through multicore parallelism. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), Broomfield, CO, USA, 6–8 October 2014; pp. 465–477.

55. Vaz Salles, M.; Cao, T.; Sowell, B.; Demers, A.; Gehrke, J.; Koch, C.; White, W. An evaluation of checkpoint recovery for massively multiplayer online games. *Proc. VLDB Endow.* **2009**, *2*, 1258–1269. [CrossRef]
56. Cao, T.; Vaz Salles, M.; Sowell, B.; Demers, A.; Gehrke, J.; Koch, C.; White, W. Fast checkpoint recovery algorithms for frequently consistent applications. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, Athens, Greece, 12–16 June 2011; pp. 265–276.
57. Kemper, A.; Neumann, T. 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, Hannover, Germany, 11–16 April 2011; pp. 195–206.
58. Shin, S.; Tirukkovalluri, S.K.; Tuck, J.; Solihin, Y. Proteus: A flexible and fast software supported hardware logging approach for nvm. In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, Boston, MA, USA, 14–17 October 2017; pp. 178–190.
59. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*, 3rd ed.; MIT Press: Cambridge, MA, USA, 2009.
60. Conover, W.J. *Practical Nonparametric Statistics*; Wiley: New York, NY, USA, 1999.
61. Knuth, D.E. *The Art of Computer Programming, Volume 3: Sorting and Searching*; Addison-Wesley: Reading, MA, USA, 1998.
62. Hellerstein, J.M.; Stonebraker, M. *Readings in Database Systems*, 5th ed.; MIT Press: Cambridge, MA, USA, 2020.
63. Yu, X.; Bezerra, G.; Pavlo, A.; Devadas, S.; Stonebraker, M. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* **2014**, *8*, 209–220. [CrossRef]
64. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing, Indianapolis, IN, USA, 10–11 June 2010; pp. 143–154.
65. TPC Benchmark C. Available online: http://www.tpc.org/tpcc/ (accessed on 23 June 2023).
66. Slaughter, J.M.; Rizzo, N.D.; Janesky, J.; Whig, R.; Mancoff, F.B.; Houssameddine, D.; Sun, J.J. High density ST-MRAM technology. In Proceedings of the 2012 International Electron Devices Meeting, San Francisco, CA, USA, 10–13 December 2012; pp. 29–33.
67. Khvalkovskiy, A.V.; Apalkov, D.; Watts, S.; Chepulskii, R.; Beach, R.S.; Ong, A.; Tang, X. Basic principles of STT-MRAM cell operation in memory arrays. *J. Phys. Appl. Phys.* **2013**, *46*, 074001. [CrossRef]
68. Liu, S.H.; Yang, W.L.; Wu, C.C.; Chao, T.S.; Ye, M.R.; Su, Y.Y.; Wang, P.Y.; Tsai, M.J. High-performance polyimide-based ReRAM for nonvolatile memory application. *IEEE Electron Device Lett.* **2012**, *34*, 123–125. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.