Index Checkpoints for Instant Recovery in In-Memory Database Systems

Leon Lee
Cloud Database Innovation Lab of HuaweiCloud
Beijing, China
liliang128@huawei.com

Siphrey Xie
Cloud Database Innovation Lab of HuaweiCloud
Beijing, China
xiexiaoqin@huawei.com

Yunus Ma
Cloud Database Innovation Lab of HuaweiCloud
Beijing, China
wenlong.ma@huawei.com

Shimin Chen
Chinese Academy of Sciences
Beijing, China
chensm@ict.ac.cn

ABSTRACT

We observe that the time bottleneck during the recovery phase of an IMDB (In-Memory DataBase system) shifts from log replaying to index rebuilding after the state-of-art techniques for instant recovery have been applied. In this paper, we investigate index checkpoints to eliminate this bottleneck. However, improper designs may lead to inconsistent index checkpoints or incur severe performance degradation. For the correctness challenge, we combine two techniques, *i.e.*, deferred deletion of index entries, and on-demand clean-up of dangling index entries after recovery, to achieve data correctness. For the efficiency challenge, we propose three wait-free index checkpoint algorithms, *i.e.*, *ChainIndex*, *MirrorIndex*, *IACoW*, for supporting efficient normal processing and fast recovery. We implement our proposed solutions in HiEngine, an IMDB being developed as part of Huawei's next-generation cloud-native database product. We evaluate the impact of index checkpoint persistence on recovery and transaction performance using two workloads (*i.e.*, TPC-C and Microbench). We analyze the pros and cons of each algorithm. Our experimental results show that HiEngine can be recovered instantly (*i.e.*, in ~10 s) with only slight (*i.e.*, 5% - 11%) performance degradation . Therefore, we strongly recommend integrating index checkpointing into IMDBs if recovery time is a crucial product metric.

PVLDB Reference Format:
Leon Lee, Siphrey Xie, Yunus Ma, and Shimin Chen. Index Checkpoints for Instant Recovery in In-Memory Database Systems. PVLDB, 15(8): 1671 - 1683, 2022.
doi:10.14778/3529337.3529350

1 INTRODUCTION

In-memory databases (IMDBs) show the promise of achieving high performance by reducing the I/O bottleneck in traditional disk-resident databases [47]. Hot topics in recent IMDB research studies include scalable index structures [2, 15, 17, 28, 33, 42], concurrency control [12, 22, 30, 40], and lightweight logging [8, 27, 45, 49]. However, the topic of *instant recovery* has received less attention, while it is an important pain point for industrial database systems. For example, the recovery time of SiloR, a well-optimized IMDB, is approximately 211s for a 70 GB TPC-C database [49]. This recovery time can hardly satisfy production requirements, especially for the increasingly popular cloud databases.

The recovery procedure of an IMDB [27, 49] typically consists of three steps: (i) loading data checkpoints, (ii) replaying logs, and (iii) rebuilding indexes. Note that Step (i) can be very fast by employing *mmap* for data checkpoints. Step (iii) does not exist in ARIES [29] (recovery mechanism for traditional disk-resident databases), while it is necessary within IMDB to recover indexes because indexes are not included in the data checkpoints or logs [27, 44, 49].

There are three categories of techniques to reduce the recovery time of IMDBs in the database literature. First, parallel log replaying techniques have been studied in depth [27, 44–46, 49]. Second, fast and frequently checkpointing [3, 5, 19, 20, 36, 49] can reduce the size of the logs that need to be replayed. Third, indexed logs [26, 37–39] support restoring tuples on demand. All these techniques target Step (ii), *i.e.*, log replaying, which is the main bottleneck of the recovery procedure. As shown in Figure 1, state-of-the-art techniques (parallel log replaying, frequent checkpoints, and indexed-log) can significantly cut down the log replaying time.

Motivation: The New Bottleneck. Existing IMDBs often recover indexes through rebuilding, *e.g.*, VoltDB [27], SiloR [49], PACMAN [44] (implemented in Peloton [31]). Figure 1 reveals that the bottleneck of the recovery procedure shifts to index rebuilding after applying the optimization techniques for log replaying. Yet the resulting recovery time is still far from the production requirement, e.g., ~10s. Index rebuilding becomes the new bottleneck!

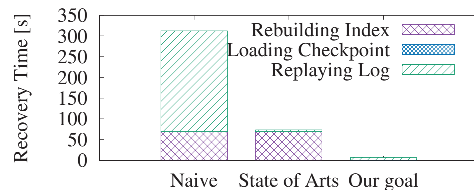


Figure 1: Rebuild index is time-consuming.

In this paper, we investigate index checkpoints to eliminate this bottleneck. Using index checkpoints, an IMDB only needs to reinsert the index entries created after the latest index checkpoint during the recovery procedure, thereby significantly reducing the amount of work in Step (iii). However, improper designs may lead to inconsistent index checkpoints or incur severe performance degradation. Therefore, the challenge is ***how to take index checkpoints in IMDBs correctly and with low overhead.***

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 15, No. 8 ISSN 2150-8097.
doi:10.14778/3529337.3529350

Correctness. As the index structure does not contain transaction information, it is almost impossible to obtain transaction consistent snapshots for an index. Thus, an index checkpoint is not transaction consistent. How to guarantee data correctness after recovery? We combine two techniques, *i.e.*, deferred deletion of index entries, and on-demand clean-up of dangling index entries after recovery, to achieve the correctness guarantee.

Efficiency. We would like to reduce the impact of index checkpoints on normal processing and achieve fast recovery time. This can be further broken into four design goals: wait-free processing, efficient index operations, fast and frequent checkpoints, and load-friendly checkpoint formats. We propose three index checkpoint algorithms to meet these goals as much as possible. (i) *ChainIndex* freezes a tree index to create a read-only snapshot for checkpointing. It maintains a list of frozen trees and uses a head tree to serve new modifications. (ii) *MirrorIndex* uses the same method to obtain index snapshots as *ChainIndex*, it keeps an up-to-date *Mirror* tree to support index reads without read amplification. (iii) *Indirection Array based CoW* (*IACoW*) employs copy on write to obtain the snapshot for checkpointing. It designs an indirection array to avoid unnecessary cascading node copies caused by path copying.

We implement our proposed solutions in HiEngine, an IMDB being developed as part of Huawei's next-generation cloud-native database product. HiEngine implements state-of-the-art features, including Adaptive Radix Tree [15, 16], Hekaton-style MVCC [12], LLAMA-alike indirection array [17, 18], Deferred Action Framework [48], the log is the database [41], indirection-array based recovery [26, 39], dataless checkpoint. We compare and evaluate the three algorithms on index checkpoints using two workloads (*i.e.*, TPC-C and Microbench) and analyze the advantages and disadvantages of each algorithm. Our experimental results show that HiEngine can be recovered instantly (*i.e.*, in ~10 s) with only slight (*i.e.*, 5% - 11%) performance degradation .

The contributions of this paper are as follows:

1. We observe that the time bottleneck of recovering an IMDB shifts from log replaying to index rebuilding. To the best of our knowledge, this paper is the first work to comprehensively study index checkpoint technology for IMDBs.
2. We investigate the correctness and efficiency challenges of index checkpoints. We combine two techniques to address the correctness issue and propose three index checkpoint algorithms, *i.e.*, *ChainIndex*, *MirrorIndex*, *IACoW*, for supporting efficient normal processing and fast recovery.
3. We evaluate and compare all algorithms within a production IMDB (*i.e.*, HiEngine) and analyze their advantages and disadvantages. Based on the experimental results, we strongly recommend that index checkpointing, especially *IACoW*, be integrated into IMDBs if the recovery time is a crucial product metric.

The structure of this paper is as follows: section 2 describes HiEngine and the challenges of index checkpoints. section 3 investigates the data correctness problem. section 4 presents the three checkpoint algorithms. section 5 reports the experimental evaluation for index checkpoints. section 6 summarizes the related work. Finally, section 8 concludes the paper.

2 BACKGROUND

In this paper, we aim to support the checkpointing of tree-based indexes in IMDBs. Without loss of generality, we discuss our solution in a concrete IMDB, HiEngine, which is being developed as part of Huawei's next-generation cloud-native database product. In this section, we briefly describe HiEngine with an emphasis on its recovery performance, then discuss the challenges for supporting wait-free index checkpointing.

2.1 HiEngine Overview

HiEngine [25] uses a variant of MVCC [12, 43] for concurrency control. It supports common isolation levels, including Read Committed, Snapshot Isolation, and Serializable. To achieve durability, HiEngine performs distributed logging [45, 49], and employs group commits [4] to minimize the impact of logging I/Os. The implementation follows the idea of "the log is the database" [41]. Basically, in the commit processing of a transaction, HiEngine serializes the *writeset* of the transaction as a batch of log records, and appends the records to the REDO log. Each log record contains the transaction's commit timestamp, the log type (e.g., insert/update/delete), the tuple ID, and a serialized representation of the entire tuple version. There is no standalone storage for the tuples in the secondary storage.

HiEngine employs Adaptive Radix Tree (ART) [15] as the main index structure. It implements Optimistic Lock Coupling [14, 16] for concurrent index accesses. As in most IMDBs, the index does not directly support the concept of versions. It is impossible to retrieve a snapshot of the index at a given timestamp. As will be discussed later, this incurs significant challenges for our design.

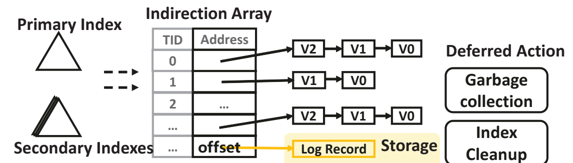


Figure 2: HiEngine architecture.

2.2 Techniques to Reduce Recovery Time

HiEngine exploits the following two techniques to achieve the recovery performance shown previously in Figure 1.

Indirection-array based recovery. HiEngine stores tuple IDs in the indexes, and creates an indirection array to map tuple IDs to the addresses of the tuples [11], as shown in Figure 2. Similar to LLAMA [18], HiEngine uses 1 bit in the 64-bit address[1] to identify whether the address is an in-memory address or an on-storage offset. In this way, the indirection array can organize memory and storage in a unified address space. During recovery processing, HiEngine re-builds the indirection array with on-storage log offsets of the latest committed tuple versions. It does not read the tuple versions during recovery. Instead, a tuple version will be loaded into memory on demand when it is used by a transaction. This

[1]Linux uses only 48 bits of the address. The upper bits are usually 0.

technique significantly reduces the amount of data to load during recovery [26, 39].

Dataless checkpoint. For each checkpoint, HiEngine obtains a consistent snapshot of data by exploiting snapshot isolation. Then it dumps the snapshot data to secondary storage. To reduce the time to load a checkpoint during recovery, HiEngine performs dataless checkpoint. That is, it does not dump the actual tuple data. Instead, it stores only the log offsets of the tuple versions in the snapshot, thereby significantly decreasing the checkpoint size. In this way, HiEngine is able to load a checkpoint quickly during recovery.

2.3 New Bottleneck and Motivation

However, since index modifications are not recorded in the log, indexes must be re-built during recovery. Unfortunately, as shown in Figure 1, this incurs significant overhead. Index rebuilding becomes the new bottleneck in the recovery phase. Similar scheme appears in existing systems, such as VoltDB [27], SiloR [49], PACMAN [44] (implemented in Peloton [31]), WBL [1], Zen [23], *etc*. For the space reason, please see our extended version on arxiv for more information on the differences between IMDB and DRDB in terms of recovery.

Motivation. First and foremost, we treat the rebuilding-base method as the evaluation baseline, and our goal in this work is to propose an effective solution to decrease the time for index recovery. Actually, this problem has not been studied in depth in the IMDB literature. If checkpoints of indexes exist during recovery, then the system can re-construct lost index entries based on the latest successful checkpoint, which is substantially faster than starting from scratch.

2.4 Challenges for Wait-Free Checkpoint

It would be nice to obtain transactionally consistent index checkpoints. However, since it is not possible to directly obtain a consistent snapshot of the indexes, a naïve approach is to block and wait. That is, when an index checkpoint request is triggered, the system blocks all the incoming transactions, and waits for all the active transactions to complete. Then, it obtains the consistent checkpoint by simply copying the latest tuple data and index entries synchronously. However, because synchronous data copying can block the system for a long time [3], this blocking approach inevitably leads to poor user experience, especially when the database size is large.

Our goal is to perform **wait-free** index checkpoints to preserve the normal transaction processing performance as much as possible. That is, we would like to take checkpoints in a non-blocking manner. Although non-blocking checkpoint algorithms for single-version databases have been studied [3, 19, 20, 34], the non-blocking algorithm for index checkpointing in IMDB has not been discussed before, which is one main contribution of this paper.

Two facts result from the wait-free design goal. First, the index checkpoint is not transaction consistent since we cannot obtain a transaction consistent snapshot of the indexes. Second, index checkpoints do not match tuple data checkpoints, since the latter is transaction consistent. Hence, we face two significant challenges for index checkpointing: 1) correctness challenge: how to guarantee data correctness after recovery? 2) performance challenge: how to design an efficient index checkpoint algorithm that minimizes the impact on transaction performance? section 3 and section 4 addresses the two challenges, respectively.

3 CORRECTNESS OF INDEX CHECKPOINTS

Timestamps used in the system. To better understand the correctness problem and our proposed solution, we first specify the timestamps used in the system:

- $globalTs$, $beginTs$ and $endTs$: HiEngine uses a global timestamp counter, $globalTs$, for generating transaction timestamps. A transaction acquires $beginTs = globalTs$ atomically when it begins. Before it starts the validation phase [12], the transaction acquires a unique $endTs$ by atomically fetching and incrementing $globalTs$.
- $minBeginTs$ and $gcTs$: $minBeginTs$ is the minimum of all active transactions' $beginTs$. Any tuple version whose end timestamp $< minBeginTs$ is not visible and can be garbage collected. Thus, $gcTs$ is periodically updated with the formula: $gcTs = minBeginTs - 1$.
- $minEndTs$ and $persistTs$: $minEndTs$ is the minimum of all active transactions' $endTs$. Hence, any transaction with $endTs$ $< minEndTs$ must have completed. If it commits successfully, then the transaction must have persisted all its modifications. $persistTs$ is periodically updated by the formula: $persistTs = minEndTs - 1$.
- $ckptTs$: Once the checkpoint request is triggered, the current $persistTs$ is assigned as $ckptTs$.

Ideal case. Ideally, both the tuple snapshot (TS) and the index snapshot (IS) obtained by the system are transaction consistent, and they are consistent with each other. As shown in Figure 3, suppose the checkpoint request is triggered at $globalTS = 200$. HiEngine executes the checkpoint operation by setting $ckptTS = persistTs = 180$. It obtains a transaction consistent TS by exploiting MVCC. Ideally, it would be nice to have a "powerful" index snapshot algorithm that can attain the IS that matches the TS. If this is the case, then during recovery, the transaction data before $ckptTS$ can be loaded from the tuple data checkpoint, and the indexes at $ckptTS$ can be loaded from the index checkpoint. After that, committed transactions with $endTs > ckptTS$ can be recovered by log replaying. Clearly, this "powerful" snapshot algorithm can guarantee the correctness of the recovered data.

Unfortunately, such a "powerful" index snapshot algorithm is infeasible. As the indexes do not contain version information, it is not possible to obtain such a transaction consistent IS. We must cope with the less ideal situation.

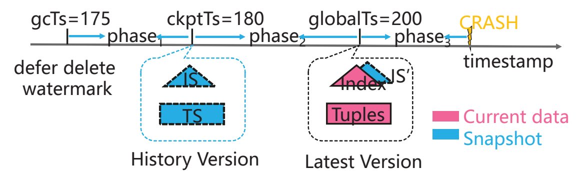


Figure 3: Ideal transaction consistent IS (left) and the realistic index snapshot IS' (right).

Problems in a realistic IS solution. As shown in Figure 3, the actual IS' taken may not be transaction consistent. However, it is straight-forward to ensure operational consistency [35, 49] such that IS' must not contain inconsistent tree nodes in the middle of any index modifications. Hence, we consider the three types of tuple write operations. 1) *update*: The system does not need to modify any indexes for tuple updates. It is sufficient to update the pointer in the indirection array to point to the new tuple version (assuming new-to-old version linked lists). Hence, we do not need to worry about tuple updates for index checkpoints. 2) *insertion*: If there is a tuple insertion in between TS and IS' ($phase_2$), then the index checkpoint (IS') will contain a dangling index entry that does not have the corresponding tuple in TS. 3) *deletion*: If a tuple is deleted in $phase_2$, the deletion may be reflected in the indexes. If the deletion is performed by an uncommitted transaction, then the tuple should exist after recovery. However, since the index entries for the tuple are missing in the index checkpoint, the tuple data can be lost because of the inconsistency.

We must correctly handle case 2) and 3). We design the following two techniques for the correctness of index checkpoints.

Deferred deletion of index entries. As shown in Figure 3, if a transaction deletes an entry during $phase_2$, the entry will not exist in IS' but it exists in TS. To remove the inconsistency because of deletion, HiEngine prevents the delete operations during $phase_2$. This is supported by the *deferred action framework* (DAF [48]). In DAF, a tuple version is deleted only when its end timestamp $\leq gcTs$. We perform the removal of index entries for tuple delete operations only when the tuple versions are garbage collected. HiEngine ensures that $gcTS$ is smaller than $ckptTs$ when the checkpoint task starts by updating the formula of $gcTS$ to $gcTs = min\{minBeginTs - 1, persistTs\}$ In this way, we can avoid missing entries in the index checkpoints.

On-demand clean-up of dangling index entries after restart. The above technique guarantees that IS' contains all the index entries in IS, but IS' may have more index entries than IS because of two reasons. (i) There are new tuple insertions during $phase_2$. If such an insertion is performed by an uncommitted transaction (which aborts[2] in $phase_3$ or is still active at the crash point), IS' contains a "dirty index entry" for the insertion. (ii) Due to the deferred execution of the delete action, the deletion in $phase_1$ is not executed. For such an index entry that is deferred to be deleted, IS' contains an "un-deleted entry". Both dirty and un-deleted entries are dangling once the system restart because they exist in the index trees but not in the tuple heap. Hence, we must carefully clean-up the dangling entries.

One way to do the cleaning is to scan all index entries and check their validity. However, this would be very expensive during recovery. Instead, we perform the clean-up with an on-demand check and delete strategy. For an index lookup, we check for two abnormal cases. First, the index lookup returns a tuple ID with an empty slot in the indirection array. Second, the key in the tuple version is different from the (prefix) key in the ART tree. In both cases, the index entry is identified as a dangling entry and is removed.

[2]The abort transaction invokes a "compensate" transaction to delete the associated tuples and index entries.

Combining the two techniques, *i.e.*, deferred deletion and on-demand clean-up, we can correctly handle both index deletions and index insertions in $phase_2$. This guarantees the correctness of index checkpoints.

4 EFFICIENCY OF INDEX CHECKPOINTS

Design goals. There are two main goals: low impact on normal processing and fast recovery time. These two goals can be further broken down into the following four design goals for a good index checkpoint algorithm:

(1) **Wait-free processing**. As discussed in subsection 2.4, the index checkpoint cannot block the normal transaction processing, and cannot introduce prominent stalls.
(2) **Efficient index operations**. The performance of the index operations should not be severely impacted because of the data structures designed for the index checkpoints. Moreover, during the checkpoint phase, the system performs batch processing for generating the checkpoint. This should not decrease the index performance too much.
(3) **Fast and frequent checkpoints**. The interval time between checkpoints determines the upper bound of the recovery time [36, 49]. Therefore, to reduce recovery time, the system should take checkpoints fast and frequently.
(4) **Load-friendly checkpoint file format**. The checkpoint file should be loaded rapidly to support instant recovery. Therefore, we decide to store checkpoint files in *mmap*-friendly layouts.

In the following, we propose three index checkpoint algorithms: *ChainIndex*, *MirrorIndex*, and *IACoW*. For each algorithm, we describe its data structure and operations, then show how it ensures durability upon crash recovery, and finally discuss the pros and cons of the algorithm.

4.1 *ChainIndex*

4.1.1 Data Structure. As shown in Figure 4, *ChainIndex* maintains a list of index trees to support the checkpoint of an index. The $headTree$ supports incoming index modifications. When a checkpoint operation is triggered, the tree will be "frozen", and regarded as an incremental snapshot of the index. This $frozenTree$ will be dumped to storage asynchronously. At the same time, a new empty tree will be initialized as the new $headTree$, which is responsible for accepting new mutate operations. After multiple checkpoints, there will be a number of $frozenTree$s on storage. The $headTree$ and the $frozenTree$s are organized into a $treeList$. Apart from $headTree$, the other trees are all read-only (*a.k.a*, frozen). Notably, $treeList$ is maintained atomically. Then, a background thread can merge the $frozenTree$s asynchronously to compute $mergedTree$.

4.1.2 Operations. We consider four main index operations: $Insert$, $Delete$, $Lookup$, and $LookupRange$. The pseudo-code of the four operations as well as the checkpoint operation is shown in Listing 1.

$Insert$. Insertion is performed on the $headTree$. Note that the index structure supports unique index keys. [3] Then the insert operation ensures that the newly inserted key does not exist in the

[3]Multiple values with the same key are handled by creating a value list as the index value.

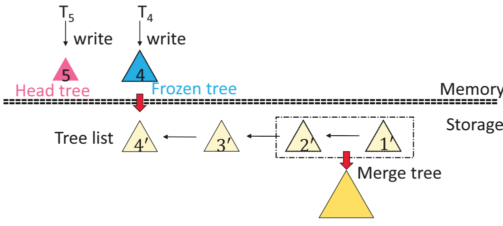


Figure 4: *ChainIndex*.

treeList (cf. Listing 1 Line 4-6). If an insert operation is ongoing when a checkpoint operation is triggered, a new *headTree* will be generated immediately (cf. Listing 1 Line 31). The ongoing insert operations are still served by the old *headTree*. The algorithm waits for these ongoing operations to complete, and serializes and freezes the old *headTree* (cf. Listing 1 Line 32,33). Insert operations that start after the trigger time of the checkpoint will be inserted into the newborn *headTree*.

Delete. The tree node related to the delete entry is marked as obsolete. Later, the memory release operation is asynchronously executed by the *Deferred Action Framework* (cf. section 3). The delete operation is straight-forward if the entry is found in the *headTree*. However, if it is found in a *frozenTree*, we mark the entry to be deleted and leave the actual removal of the entry in the merge phase (cf. subsubsection 4.1.5).

Lookup. The lookup first searches the current *headTree*. If the entry is found in *headTree*, it returns directly. If not, the algorithm will look for the entry tree-by-tree until the entry is found. Note that this search process can lead to significant read amplification.

LookupRange. Unlike *Lookup*, *LookupRange* can have multiple results. It also needs to find results tree-by-tree. The difference is that we need to merge all the trees' results and sort the results in ascending or descending order. A *mergeSort*-like algorithm is suitable since each tree's results are sorted already.

4.1.3 Running Example. Figure 5 shows an example to illustrate how *ChainIndex* works. ① Initially, there are three *frozenTrees* and one *headTree* (*i.e.*, $tree_4$). Three operations target at the *headTree*. I_1 inserts an entry into $tree_4$. D_1 marks the corresponding node as obsolete, which will be released by *DAF*. Note that if the delete entry is not in $tree_4$, it should be cleaned during the merge phase. L_1 looks up a unique entry from the *treeList* (*i.e.*, $\{tree_4, tree'_3, tree'_2, tree'_1\}$). ② The checkpoint processing is triggered at time T_4. The index immediately and atomically creates a new *headTree* (*i.e.*, $tree_5$) to serve new index modifications. ③ Suppose three operations are ongoing at time T_4. I_2 still inserts into $tree_4$ though it is not the *headTree* any more. D_2 marks the corresponding node as obsolete. However, since $tree_4$ becomes a *frozenTree*, the deleted nodes (if committed) will be cleaned up during the merge phase. L_2 still finds result in the *treeList* $\{tree_4, tree'_3, tree'_2, tree'_1\}$. ④ After the ongoing modification operations (I_2 and D_2) complete (at T'_4), the checkpoint thread asynchronously serializes and dumps the read-only $tree_4$ into storage. ⑤ Then, any index modifications (e.g., I_3 and D_3) that come after the checkpoint time are served by the new *headTree*. L_3 will look for the entry in $tree_5$ first. If the entry is found, it returns directly, otherwise it will check the frozen $tree'_4$, ..., $tree'_1$.

```
1  bool ChainIndex::Insert(KEY key, uint64_t TID) {
2      headTree = getHead(); // atomically.
3      // whether the pair <key,TID> exists?
4      for tree = headTree to getTail() {
5          if(tree.exist(key)) return false;
6      }
7      return headTree.Insert(key, TID); // insert <key,TID> into the headTree.
8  }
9  bool ChainIndex::Delete(KEY key, uint64_t TID) { // invoked by DAF.
10     headTree = getHead();
11     headTree.Delete(key,TID); // Note: the key may not exist in headTree.
12 }
13 TID ChainIndex::Lookup(KEY key) {
14     headTree = getHead; TID = 0;
15     for tree = headTree to getTail() { // find the entry tree-by-tree
16         if(TID = tree.Lookup(key))
17             return TID;
18     }
19     return 0; // TID = 0 means not found.
20 }
21 ResultSet ChainIndex::LookupRange(KEY begin, KEY end, uint64_t count) {
22     headTree = getHead();
23     ResultSet = empty;
24     for tree = headTree to getTail()
25         ResultSet.insert(tree.LookupRange(begin,end,count));
26     MergeSort(ResultSet);
27     return ResultSet.SubSet(0,count-1);
28 }
29 void ChainIndex::Checkpoint() {
30     headTree = getHead();
31     insertTreeList(new Tree()); // atomically insert a newborned headTree.
32     Wait(); // until all the active modifying operations finished.
33     headTree.serialize(); // headTree is readonly.
34 }
```

Listing 1: Algorithm of *ChainIndex*.

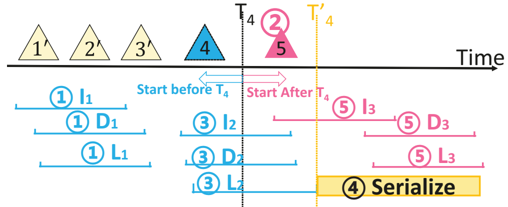


Figure 5: Running example for *ChainIndex*.

4.1.4 Serialize and Recover Index Trees. As index structures are pointer based, we adopt *pointer swizzling* [13] to persist the pointers.

Unswizzling. Before serializing and saving a tree node into storage, we should first convert the child pointers into file offsets (*a.k.a*, *unswizzling*). To make things easier, we traverse the tree in *post-order* and persist the tree node-by-node. *Post-order* guarantees that children's file offsets can be calculated before the parent node is persisted. The root node of the tree will be stored at the tail of the checkpoint file.

Swizzling. During the recovery phase, we first *mmap* the checkpoint file, then the pointer address of a node can be calculated with the following formula: $memPointer = mmapAddress + offset$, where *mmapAddress* represents the address of the checkpoint file's mmap address, *offset* is the address of the node in the checkpoint file, and *memPointer* is the memory address of the node.

4.1.5 Tree Merging and Garbage Collection. After multiple checkpoints, the *treeList* will become longer and longer. This introduces two problems. First, the read amplification becomes more and more

severe. Second, the longer the list, the more the garbage and wasted space, since the garbage in $frozenTree$s have not been cleaned.

Therefore, an additional thread needs to be introduced to periodically merge the $frozenTrees$ and piggyback the deletion of garbage entries. For example, a $treeList$ consists of five trees: $tree$ 5, $tree$ 4′,$tree$ 3′, $tree$ 2′, $tree$ 1′. Then, a merge of { $tree$ 1′,$tree$ 2′ } generates a $mergedTree$, and at the same time, releases relevant garbage (*a.k.a*, dangling) entries.

4.1.6 Summary. **Pros**: *ChainIndex* achieves design goal (1), (3), and (4). It supports wait-free processing, incremental checkpoints, and load-friendly file format. The checkpoints are incremental as a $frozenTree$ records the modifications in a period of time.

Cons: *ChainIndex* incurs several problems. (i) One explicit issue is the *read amplification* introduced by $treeList$. Note that this exists not only in the lookup operation, but also in the insert operation, as it needs to check whether the entry exists in the $treeList$ before inserting an entry. (ii) Garbage nodes in the $frozenTrees$ can only be cleaned during the merge phase. (iii) The post-order traversal for generating serialized checkpoints is very time-consuming.

4.2 *MirrorIndex*

We would like to reduce the read amplification of *ChainIndex*. Inspired by Wait Free Ping-Pong [3] algorithm for data checkpoints, we propose a *MirrorIndex* algorithm.

4.2.1 Data Structure. To solve the *read amplification* problem, *MirrorIndex* enhances *ChainIndex* with an extra *Mirror* tree for index reads, as shown in Figure 6. The *Mirror* tree is a “full” tree, containing all the index entries. Therefore, read operations can obtain results through the *Mirror* tree directly. *MirrorIndex* does not need to traverse $treeList$ any more, avoiding the read amplification due to $treeList$. On the other hand, each index entry has to be inserted into both the *Mirror* tree and the $headTree$.

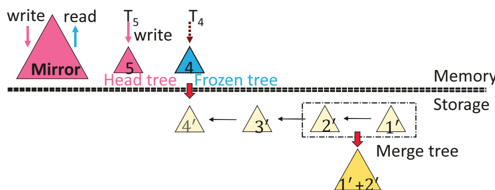


Figure 6: *MirrorIndex*.

4.2.2 Operations. We discuss the operations during normal processing. Special operations during recovery will be presented in subsubsection 4.2.4.

$Insert$. Compared with *ChainIndex*, *MirrorIndex* needs a dual insert operation. For an entry to insert, *MirrorIndex* first inserts the entry into the *Mirror* tree. Then it inserts the entry into the $headTree$ if and only if the insertion to the *Mirror* tree is successful (cf. Listing 2 Line 3-5). Note that *MirrorIndex* does not need to check whether the entry exists in the $treeList$ before inserting it into $headTree$.

$Delete$. To delete an entry, *MirrorIndex* has to delete it from both the *Mirror* tree and the $headTree$. Note that the entry may not exist in the $headTree$. In such cases, the entry is stored in a $frozenTree$ of the $treeList$, and will be deleted in the merge phase.

Lookup & LookupRange. Both point and range queries can get results directly from the *Mirror* tree (Line 19,25). Compared with *ChainIndex*, *MirrorIndex* eliminates the read amplification.

$Checkpoint$. *MirrorIndex* follows the same procedure as *ChainIndex* to serialize and save a $frozenTree$ to persistent storage (cf. subsubsection 4.1.4).

```
1  bool MirrorIndex::Insert(KEY key, uint64_t TID) { // Dula insert.
2      headTree = getHead(); // atomically.
3      if (mirror.recovered) { // normal case.
4          if (mirror.Insert(key,TID)) {
5              headTree.Insert(key,TID); return true;}
6      } else { // occurs during step 3 in recovery phase.
7          if(headTree.Insert(key,TID)) {
8              mirror.Insert(key,TID); return true;}
9      }
10     return false;
11 }
12 bool MirrorIndex::Delete(KEY key, uint64_t TID) { // Dual delete.
13     headTree = getHead();
14     mirror.Delete(key,TID);
15     headTree.Delete(key,TID); // The key may not exist in headTree.
16 }
17 TID MirrorIndex::Lookup(KEY key) {
18     if(mirror.recovered) {
19         return mirror.Lookup(key);
20     } else  // occurs during step 3 in recovery phase.
21         return treeList.Lookup(key);
22 }
23 ResultSet MirrorIndex::LookupRange(KEY begin, KEY end, uint64_t count) {
24     if(mirror.recovered) {
25         return mirror.LookupRange(begin,end,count);
26     } else  // occurs during step 3 in recovery phase.
27         return treeList.LookupRange(begin,end,count);
28 }
29 void MirrorIndex::Checkpoint() {
30     headTree = getHead();
31     insertTree(new Tree()); // atomically insert to head of the list.
32     Wait(); // until all the active modifying operations finished.
33     headTree.serialize(); // headTree is readonly.
34 }
35 void MirrorIndex::Recovery() {
36     mirror.recovered = false;
37     treeList.mmap();     // step 1.
38     headTree = new Tree();
39     headTree.Recovery();  // step 2, the system is available after step 2.
40     Thread.run(mirror.Recovery()); // step 3.
41 }
```

Listing 2: Algorithm of *MirrorIndex*.

4.2.3 Running Example. We reuse Figure 5 to show the running example of *MirrorIndex*. ① Initially, there are three operations. I_1 inserts entries into both the *Mirror* tree and $tree_4$. D_1 marks the corresponding nodes as obsolete, and *DAF* will release those nodes. Note that both the entry in the *Mirror* tree and $tree_4$ should be deleted. Unlike $ChainIndex$, L_1 looks up a unique entry directly in the *Mirror* tree. ② The checkpoint processing is triggered at T_4. The index immediately generates a new tree (*a.k.a*, $tree_5$) to accept new modifications. ③ There are three ongoing operations at T_4. I_2 still inserts into both the *Mirror* tree and $tree_4$. ($tree_4$ is not the $headTree$ any more). D_2 marks the corresponding nodes as obsolete. If the transaction commits, those nodes in $tree_4$ will be cleaned up in the merge phase. L_2 still finds results in the *Mirror* tree directly. ④ After the active modification operations (*i.e.*, I_2 and D_2) complete at T_4', the checkpoint thread serializes the read-only $tree_4$ into storage. ⑤ I_3 inserts an entry into the *Mirror* tree and $tree_5$, and D_3 deletes the entry from both the *Mirror* tree and $tree_5$.

If the entry does not exist in $tree_5$, then the entry should be deleted during the merge phase. L_3 finds the entry in the *Mirror* tree.

4.2.4 Recovery. The recovery procedure consists of three steps:

(1) *Step 1: mmap frozen trees.* The system first *mmap*s the index checkpoint files (*i.e.*, persistent trees) and organizes the trees into a $treeList$.
(2) *Step 2: recover headTree.* The system rebuilds the $headTree$ (cf. $headTree$ includes all the entries generated in $phase_3$).
(3) *Step 3: rebuild Mirror tree in the background.* The system is available after Step 2. It can serve new requests using the $treeList$. In the meantime, the system rebuilds the *Mirror* tree in the background.

Importantly, step 3 can support new requests since the $headTree$ has been recovered. Note that before the *Mirror* tree is recovered, the operations are different from the normal case. For $Insert$ operations, we should first insert the entry into the $treeList$, then insert it into the *Mirror* (cf. Listing 2 lines 6-9). For $Lookup$ and $LookupRange$ operations, we should traverse the $treeList$ tree-by-tree (cf. Listing 2 lines 21,27), as in *ChainIndex*.

4.2.5 Summary. **Pros**: *MirrorIndex* inherits most strengths of *Chain-Index*. In addition, it eliminates *read amplification* by searching the *Mirror* tree.

Cons: One potential issue is the *write amplification* introduced by "dual insert/delete". Moreover, *MirrorIndex* still employs the expensive post-order traversal for serializing $frozenTree$s.

4.3 *IACoW*

The above designs (*i.e.*, *ChainIndex* and *MirrorIndex*) create a list of $frozenTree$s for supporting incremental checkpoints. An alternative approach is to exploit the idea of $Copy\ on\ Write$. The typical implementation of $Copy\ on\ Write$ for tree structures is $Path\ Copying$. As shown in Figure 7, when a tree node D is modified, $Path\ Copying$ first makes a "backup" copy of the node, then modifies the node in the tree. The resulting node is D'. However, it entails that the child pointer in D's parent node B must reflect the change. This leads to cascading $Copy\ on\ Write$ of node B and then node A. In short, the scheme copies the root-to-leaf path for modifying node D.

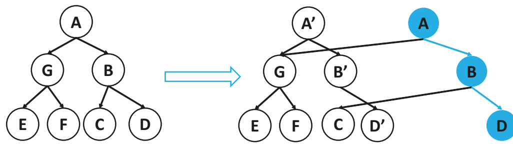


Figure 7: Path copying for a tree structure.

To eliminate the unnecessary copy of ancestor nodes, we exploit an indirection array to store the mapping from node ids to node addresses. We propose the $Indirect\ Array\ based\ Copy\ on\ Write$ (*IACoW*) algorithm, which only needs to copy modified tree nodes during the checkpoint phase.

4.3.1 Data Structure. (1) $Indirection\ Array$. As shown in Figure 8, we assign a node id (*a.k.a*, logical pointer) to each tree node. The id of the root node is fixed and set to 0. The indirection array $NID[...]$ stores the mapping from node ids to node pointers. Then, a child pointer in a tree node is not a memory address any more. Instead, it stores the node id of the child node. Given the child $nodeid$, the child node can be retrieved with $NID[nodeid]$.

(2) $Epoch$. An epoch is the time interval between two subsequent index checkpoints. First, we maintain a global epoch counter, $globalEpoch$, for each tree. At the start of every index checkpoint, we set the $ckptEpoch$ to be the current $globalEpoch$, and monotonically increment $globalEpoch$ (cf. Listing 3 Line 11). The system guarantees that there is only one ongoing checkpoint operation on an index. Hence, $ckptEpoch \equiv globalEpoch - 1$. Second, every tree node maintains an $Epoch$ field. $node.Epoch$ identifies the epoch when the node is last modified (cf. Listing 3 Line 4). Third, a thread sets its $threadEpoch$ to be the current $globalEpoch$ before performing any index operations (cf. Listing 3 Line 7).

(3) $Multi\ version\ tree\ node$. As shown in Figure 8, we avoid $Path\ Copying$ for performing $Copy\ on\ Write$ for node D. We create a new version D' and let D' point to D. Then we atomically change $NID[3]$ to point to D'. In this way, the parent node B can use the unchanged node id to locate the child node. The ancestors do not need to be copied. To reduce the space overhead, we ensure that there is at most one version per epoch for a given node. That is, multiple modifications within the same epoch will be performed on the same version of the node. Moreover, we actively garbage collect obsolete node versions. If a node has two versions $n1$ and $n2$ such that $n1.Epoch < n2.Epoch < globalEpoch$, then $n1$ is obsolete.

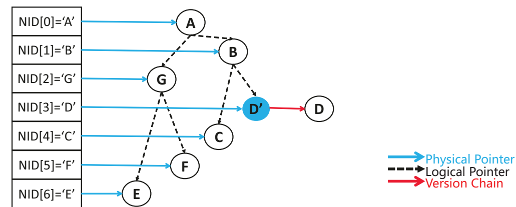


Figure 8: *IACoW*.

4.3.2 Tree Operations. The operations of *IACoW* are similar to the original ART [15, 16] except that (i) it needs to use the node ids (logical pointers) (cf. Listing 3 Line 3) and (ii) perform epoch-based node management. We also use $Optimistic\ Lock\ Coupling$ [14, 16] for concurrent index accesses.

4.3.3 Epoch-based tree node management. Listing 4 shows the operations for epoch-based tree node management.

$NewNode$. When we allocate a new node due to an insert operation, we need to get an unused node id. Then we allocate space for the node and insert it into the indirection array. The $node.Epoch$ is set to $threadEpoch$.

$GetNode$. The tree operation thread always reads the latest version of the node, which is given by the indirection array NID_array.

$isNeedCoW$. Before modifying a node, we call $isNeedCoW$ to determine whether $Copy\ On\ Write$ should be done. If $threadEpoch < globalEpoch$, then $threadEpoch = ckptEpoch = globalEpoch - 1$. Index checkpoint is triggered and it will wait for this thread to complete the index operation. The modification can go to the current node without copying because the node's $Epoch$ will be set to $threadEpoch$ (*i.e.*, $ckptEpoch$) and the node will be correctly

```
class N4 : public Node { // N16, N48, N256 also use logical pointer.
    Key keys[4];
    NID children[4]; // logical pointers.
    uint64_t Epoch; // inline epoch.
};
bool IACoW::Insert(KEY key, uint64_t TID) { // Other operations are omitted.
    threadEpoch = gloablEpoch;
    return ART_OLC::Insert(key, TID, threadEpoch);
}
void IACoW::Checkpoint() {
    ckptEpoch = globalEpoch++;
    Wait(); // until all the active tree operations finished.
    NodeManager::serialize(ckptEpoch); // scan version based on epoch.
    // all the garbage entries should be released until checkpoint finished.
    releaseSnapshotVersion(ckptEpoch);
}
void IACoW::Recovery() {
    Load ckptEpoch;
    recover Indirection Array;
    mmap nodeFile;
    globalEpoch = ckptEpoch + 1;
}
```

Listing 3: Algorithm of epoch-based *IACoW*.

saved in the checkpoint. Otherwise, there is no on-going checkpoint operation. If $threadEpoch = node.Epoch$, then the node has been modified in the same epoch before. Since we keep a single version per epoch, we do not copy the node. Only when $threadEpoch \neq node.Epoch$ does $isNeedCoW$ return true.

$UpdateNode$. We check $isNeedCoW$ before updating a node. If it returns false, the node is updated in place. If it returns true, a new version of the node is created and updated. Note that the node's $Epoch$ is set to $threadEpoch$. We copy four parts of information: node type, entry data, pointer (a.k.a, NID or TID), lock state[4].

$DeleteNode$. DeleteNode is a special kind of $UpdateNode$. It marks the node as obsolete, which is later released by DAF.

$GrowNode$ & $ShrinkNode$. ART has four different node sizes. $GrowNode$ ($ShrinkNode$) allocates a bigger (smaller) node and copies the data of the old node to the new node. If $isNeedCoW$ returns false, it replaces the old node with the new node, and the old node is handed over to DAF for recycling. If $isNeedCoW$ returns true, then the new node should be inserted to the head of the version chain. Note that the version of the new node is different from the old node.

4.3.4 Parallel Checkpoint. Unlike *ChainIndex* and *MirrorIndex*, there is no need to serialize the tree structure by post-order traversal for *IACoW* (cf. subsubsection 4.1.4). First, we can scan the indirection array NID_array in parallel, get the latest node versions that satisfy the checkpoint condition (specified below) and serialize them to storage. Second, we persist the corresponding indirection array entries. We replace a node's memory address with its offset on storage. Then the indirection array maintains the mapping between the logical node ids and the physical offsets. Finally, we save the current $ckptEpoch$. After the checkpoint finishes, the snapshot garbage can be deleted (cf. Listing 3 Line 15 and Listing 4 Line 25,50).

$Checkpoint\ condition$. There are two ways to set the condition. (i) $node.Epoch = ckptEpoch$: This will create an incremental checkpoint. (ii) $node.Epoch \leq ckptEpoch$: This will create a full checkpoint of the index. We use (i) to reduce the checkpoint cost. However, if all checkpoints are incremental, the recovery time will be large.

[4]We do not strictly distinguish between lock and latch here.

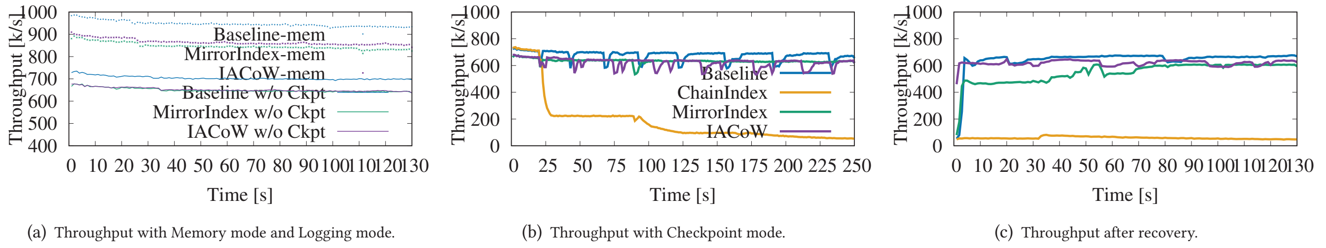
```
Node *NodeManager::NewNode(uint64_t threadEpoch) {
    Node *newNode = new Node(threadEpoch);
    freeid = getFreeid();  // the id should be recycleable.
    NID_Array[freeid]->atomicInsertVersion(newNode);
    return newNode;
}
Node *NodeManager::getNode(NID id) {
    return NID_Array[id];  // always return the latest version of the node.
}
bool NodeManager::isNeedCow(uint64_t threadEpoch, Node* node) {
    if (threadEpoch < globalEpoch) // tree operations start before checkpoint
        return false;
    if (threadEpoch == node->Epoch) // the node has been copied before
        return false;
    return true;
}
void NodeManager::UpdateNode(uint64_t threadEpoch, NID id, Byte byte) {
    Node *oldNode = getNode(id);
    if (!isNeedCoW(threadEpoch, oldNode)) {
        oldNode->Update(byte, threadEpoch);
    } else {
        newNode = oldNode->copy();  // copy to a new node.
        newNode->Update(byte, threadEpoch);
        NID_Array[id]->atomicInsertVersion(newNode);
        snapshot_garbages->add(oldNode); // delete after checkpoint finished.
    }
}
void NodeManager::DeleteNode(uint64_t threadEpoch, NID id) {
    Node *oldNode = getNode(id);
    if (!isNeedCoW(threadEpoch, oldNode)) {
        oldNode->obsolete = true;
        garbages->add(oldNode);  // delete after the transaction commit.
    } else {
        newNode = oldNode->copy();  // copy to a new node.
        newNode->obsolete = true;
        NID_Array[id]->atomicInsertVersion(newNode);
        garbages->add(newNode);  // delete after the transaction commit.
    }
}
// shrinkNode is silimar with growNode
void NodeManager::growNode(uint64_t threadEpoch, NID id, Byte byte) {
    Node *oldNode = getNode(id);
    Node *biggerNode = oldNode->CopyToBiggerNode();
    biggerNode->Update(byte, threadEpoch); // update in the bigger node.
    if (!isNeedCoW(threadEpoch, oldNode)) {
        NID_Array[id]->atomicReplaceVersion(biggerNode);
        garbages->add(oldNode);  // delete after the transaction commit by DAF.
    } else {
        NID_Array[id]->atomicInsertVersion(biggerNode);
        snapshot_garbages->add(oldNode); // delete after checkpoint finished.
    }
}
```

Listing 4: Algorithm of Epoch-based NodeManager.

Hence, we periodically employ (ii) so that recovery can start from the full checkpoint.

4.3.5 Running Example. We also use Figure 5 to show how *IACoW* works. Suppose that the current $globalEpoch = 3$. ① I_1, D_1, L_1 all have $threadEpoch = 3$. When an operation creates a new node, the corresponding node's $Epoch$ is set to 3. $isNeedCoW$ is checked before updating a node. It returns false, meaning that copy on write must not be performed. ② A checkpoint task is triggered at T_4. $ckptEpoch = globalEpoch = 3$ then $globalEpoch$ is incremented to 4. ③ The three ongoing operations at checkpoint time, *i.e.*, I_2, D_2, L_2, get $threadEpoch = 3$. Hence, their behavior is the same as I_1, D_1, L_1. ④ After active modifications (*i.e.*, I_2 and D_2) finish at T_4', the checkpoint can serialize the snapshot ($ckptEpoch = 3$) into storage. ⑤ I_3, D_3, L_3 all get $threadEpoch = 4$. $isNeedCoW$ returns true for I_3, D_3, thus copy on writes are performed.

4.3.6 Recovery. We *mmap* the latest full index checkpoint and the incremental index checkpoints after the full checkpoint to memory during the recovery phase. We load the indirection array from the



(a) Throughput with Memory mode and Logging mode.

(b) Throughput with Checkpoint mode.

(c) Throughput after recovery.

Figure 9: Time-series of throughput.

full checkpoint and update it with the entries from the incremental checkpoints. Note that the indirection array points to the nodes in the read-only *mmap*-ed files. Therefore, the modification of a node will trigger the *CoW* action. We set *globalEpoch* to one plus the largest *ckptEpoch* recorded in the checkpoints at the end of the recovery phase (cf. Listing 3 Line 21).

4.3.7 Summary. **Pros**: (i) Since *IACoW* has a single tree structure, we expect its read performance to be better than *ChainIndex* and *MirrorIndex*. (ii) The checkpoint can be efficient because it avoids the costly post-order tree traversal.

Cons: The design of the "logical pointer" through the indirection array leads to read amplification (*i.e.*, more memory references for an index operation).

5 EVALUATION

5.1 Experimental Setup

5.1.1 Infrastructures. We deployed the system on a 112-core server with 4 Intel(R) Xeon(R) Platinum 8280L CPUs and 1500 GB of DRAM. Besides, each socket equipped with 744 GB NVM, plugged into NVDIMM interleaved with DRAM, *i.e.*, 3 TB of NVM in total. As HiEngine needs to be frequently appended writing and random read storage device, we write logs and checkpoint data to NVM devices to minimize the impact of I/O. Moreover, NVM adopts the *Appdirect* mode and installs an XFS file system. At the same time, a 2 MB huge page configuration is used [32]. We only evaluate the performance in a single NUMA socket to avoid the phenomenon that the performance of NVM and DRAM is significantly degraded when accessing remote NUMA sockets. The OS is Linux 4.18 x86_64, the optimized configuration of GCC 7.3 with (-O3) is adopted. Besides those proposed schemes, we also implement *Baseline* scheme which only takes data checkpoints but does not take index checkpoints and indexes are recovered by rebuilding.

5.1.2 Checkpoint configurations. HiEngine allows us to compare the following configuration mode as a plug-gable component.

- **Memory mode**. The system disables logging and checkpoint, which does not support recovery. All of them are labeled with "*Baseline-mem*", "*ChainIndex-mem*", "*MirrorIndex-mem*", "*IACoW-mem*";
- **Logging mode**. In this mode, the logging must flush to the NVM devices before the transaction commit. However, it disables the checkpoint. We label it with "*Baseline w/o ckpt*", "*ChainIndex w/o ckpt*", "*MirrorIndex w/o ckpt*", "*IACoW w/o ckpt*";
- **Checkpoint mode**. Besides logging, the system also triggered checkpoint operations periodically. It labeled with "*Baseline*", "*ChainIndex*", "*MirrorIndex*", "*IACoW*".

5.1.3 Workloads. We use two benchmarks to evaluate the performance of each algorithm.

TPC-C. TPC-C is the de-facto industry standard to evaluate OLTP systems. It contains nine tables that simulate an online order processing application. The initial data size for the workload is ~100MB per warehouse. We also implement the standard proportions of all the five mix transactions, *i.e.*, Neworder 45%, Payment 43%, Ordstat 4%, Delivery 4%, StockLevel 4%.

Microbench. To evaluate and analyze some particular scenarios, we implement a variant workload based on TPC-C, named *Microbench*. Only the orderline table of TPC-C is reserved in Microbench, and the default data contains 1,000,000 rows. Four types of transactions are executed upon this table: (i) *Insert*: insert ten tuples; (ii) *PointQuery*: find a random tuple for ten times; (iii) *RangeQuery*: range scan and return top-10 rows in this range; (iv) *Delete*: randomly delete an existing row. Microbench does not have update transactions for two reasons: First, the update operation does not modify the index (cf. subsection 2.1), which can pay more attention to the performance comparison of the index itself. Second, No update means each row has only one version, and it can minimize the impact of MVCC-based transaction processing. By adjusting the proportion of different transactions, we can simulate the following four scenarios:

- **Write Heavy**. 80% of *Insert* transactions, 10% of *PointQuery* transactions, and 10% of *RangeQuery* transactions.
- **Read Heavy**. 20% of *Insert* transactions, 40% of *PointQuery* transactions, and 40% of *RangeQuery* transactions.
- **Middle**. 30% of *Insert* transactions, 30% of *PointQuery* transactions, and 40% of *RangeQuery* transactions.
- **TPCC-alike**. 30% of *Insert* transactions, 30% of *PointQuery* transactions, 30% of *RangeQuery* transactions, and 10% of *Delete* transactions.

5.2 Performance Evaluation

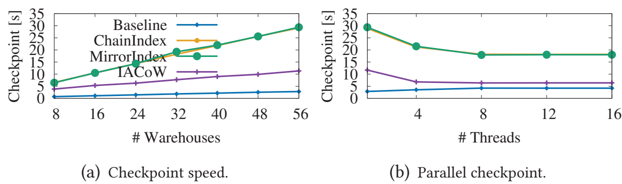
5.2.1 Prominent stalls. Figure 9 shows the log, checkpoint, and recovery impact on TPC-C's performance with 16 threads and 16 warehouses. Figure 9(a) shows that the performance of each algorithm, both memory mode and logging mode is relatively stable.

However, the performance of logging mode is decreased by about 20% for each algorithm, which is caused by I/O.

As shown in Figure 9(b), all algorithms take multi-round of checkpoints. To be fair, apart from the first-round checkpoint, all the following checkpoints are incremental. We can get the following findings with checkpoint mode: (i) Since *Baseline* only taking tuple checkpoint, the performance is degraded a little during the checkpoint phase but is soon back to its previous level after the checkpoint phase. (ii) *ChainIndex* leads to performance drop precipitously, which is caused by reading amplification. (iii) Compared with *Baseline*, *MirrorIndex* does not introduce an additional spike, and the relative performance of *MirrorIndex* is 9% lower than *Baseline*. However, checkpoint tasks for *MirrorIndex* take a long time for the reason of post-order traversal. (iv) The performance curves of *IACoW* and *Baseline* have a similar pattern, but the relative performance of *IACoW* is 9% lower than *Baseline*.

Figure 9(c) shows the performance of the system after recovery. Note that the figure does not involve the recovery phase. The corresponding recovery time is shown in Figure 11(a). It can be seen that after the system is available, *Baseline*, *ChainIndex*, and *IACoW* can quickly reach the performance of the system before the crash. However, the performance of *MirrorIndex* gradually increases and reaches a stable level at approximately 50s, as the background thread needs almost 50s to recovery *Mirror* trees (cf. Step 3 of subsubsection 4.2.4).

5.2.2 Checkpoint speed. Figure 10(a) shows the checkpoint duration time when they all use a single thread to take checkpoints. It can be found that (i) the larger the data size, the longer the checkpoint time. (ii) *IACoW* and *Baseline* are faster than *ChainIndex* and *MirrorIndex*. (iii) Significantly, the checkpoint time of *ChainIndex* and *MirrorIndex* reaches 30 s if warehouse=56. Thus *ChainIndex* and *MirrorIndex* are not suitable for large dataset scenarios.



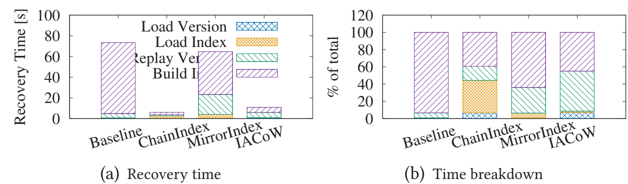
(a) Checkpoint speed. (b) Parallel checkpoint.

Figure 10: Checkpoint.

5.2.3 Parallel index checkpoint. **Intra or Inter**? The checkpoint procedure should be parallelized to support fast and frequent checkpoints. *IACoW* adopts the intra tree parallelism, as the tree structure can be divided easily with the help of the indirection array. Technically, *ChainIndex* and *MirrorIndex* can split the tree into multiple subtrees and perform post-order traversal for each subtree to support intra tree parallelism. However, ART is not a balanced tree, which will lead to the problem of uneven parallelism. Therefore, for simplicity, *ChainIndex* and *MirrorIndex* adopt the inter tree parallelism.

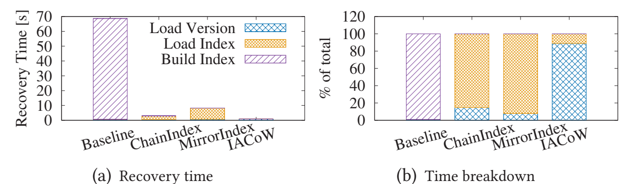
Figure 10(b) gives the relationship between checkpoint time and checkpoint thread number when warehouse=56. (i) Checkpoint time decreases as the number of threads increases; (ii) *Baseline* and *IACoW* can bound the time of the checkpoint within 5 s; (iii) In addition, when the number of threads is greater than 8, the checkpoint time of *ChainIndex* and *MirrorIndex* is not improved. Because TPC-C has 10 indexes, and each thread is responsible for taking checkpoints for a single tree, up to 10 threads are used for parallelization. (iv) The checkpoint duration time of *IACoW* is not improved when the number of background threads is more than 4 because *IACoW*'s parallel checkpoint algorithm is heavy fast. The bandwidth of the I/O device is the main bottleneck.

5.2.4 Recovery. **Unplanned crash.** Figure 11 shows the comparison of recovery time of different algorithms in case of an unplanned crash. We use the SIGKILL signal to simulate the scene of a crash. Figure 11(a) gives the corresponding recovery time of Figure 9(c). We can find that (i) *Baseline* still spends many time building indexes; (ii) The reason why *ChainIndex* recovers quickly is that the performance of the *ChainIndex* algorithm is slow, and the size of logs to be replayed and the size of checkpoint file are both small; (iii) Since the checkpoint frequency of *MirrorIndex* is very slow, it takes a long time to recover; (iv) *IACoW* can recover quickly within 10s, which benefits from the frequent checkpoint strategy.



(a) Recovery time (b) Time breakdown

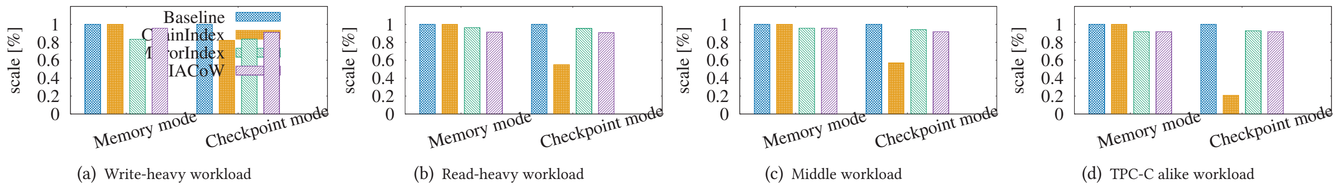
Figure 11: Recovery with unplanned crash.



(a) Recovery time (b) Time breakdown

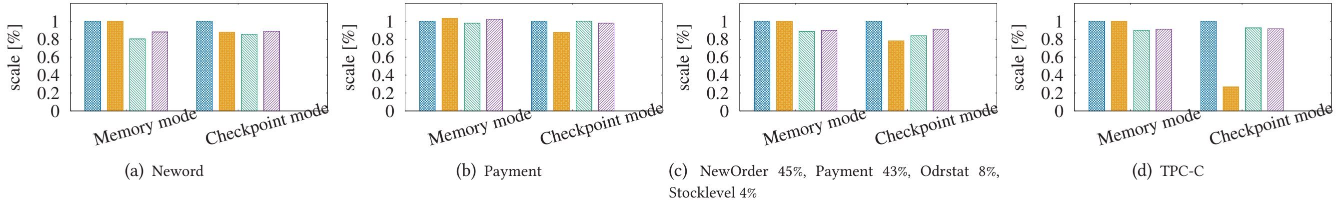
Figure 12: Recovery with planned shutdown.

Planned shutdown. Figure 12(a) shows the effect of data volume on index recovery time with a planned shutdown. Before the system shutdown, the system manually executes the planned checkpoint once. Therefore, during the recovery phase, the system does not need to replay the log but only load the checkpoint. As a result, we can find that: (i) the recovery time of *Baseline* is much longer than that of the other three schemes because the wholly rebuild index is time-consuming; (ii) *ChainIndex*, *MirrorIndex*, and *IACoW* can be instantly recovered by loading checkpoint data; (iii) *MirrorIndex* is slower than *IACoW* since the data structure for *IACoW* is much more friendly to loading; (iv) *ChainIndex* is faster than *MirrorIndex* since the checkpoint data size is much smaller than *MirrorIndex*.



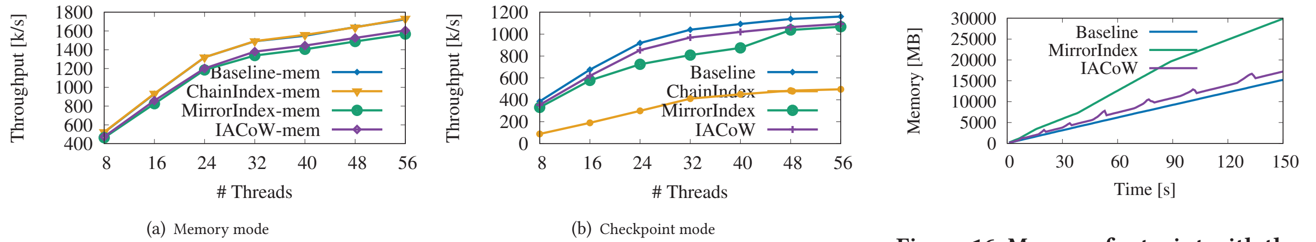
(a) Write-heavy workload (b) Read-heavy workload (c) Middle workload (d) TPC-C alike workload

Figure 13: Throughput on Microbench.



(a) Neword (b) Payment (c) NewOrder 45%, Payment 43%, Odrstat 8%, Stocklevel 4% (d) TPC-C

Figure 14: Throughput on TPC-C.



(a) Memory mode (b) Checkpoint mode

Figure 15: Scalable on TPC-C and memory footprint with the same throughput.

Figure 16: Memory footprint with the same throughput.

5.2.5 Throughput comparison. All the algorithms have modified the structure of ART, which inevitably causes some performance losses. This part shows the detail about those losses.

Vary workloads. Figure 13 shows the performance of Microbench with different scenarios, y-axis shows the performance ratio relative to *Baseline*. The initial table size of Orderline contains 1,000,000 rows of data. We can find that: (i) *Baseline* performs best because it does not need any extra operations on ART. (ii) *ChainIndex* has the same performance as *Baseline* in memory mode because *ChainIndex* does not execute checkpoint, which is similar to *Baseline*. However, the performance of *ChainIndex* is reduced by 45% in read-heavy for reading amplification and 79% in TPC-C alike scenario for $frozenTree$ containing many garbage entries. (iii) The performance of *MirrorIndex* decreases by 22% in the case of heavy write workload, but only by 5% in the case of trivial read workload. (iv) The performance of *IACoW* is relatively stable in all workload scenarios. Compared with *Baseline*, the performance loss is 5%-11%. The decrease is due to an additional pointer chasing through the direction array when reading ART's node.

Since Microbench's transaction pattern is simple, the execution cost for transactions is negligible. To combine transaction and log modules for end-to-end performance testing, we also analyze the workload of TPC-C. Figure 14(a) and Figure 14(b) show the throughput of Neworder and Payment transaction respectively, the result is similar to that of Figure 13(a). Compared with Figure 14(d), there is no delivery transaction in Figure 14(c) because only the delivery transaction contains delete operations, the performance of *ChainIndex* is similar to that of Figure 13(d).

Scalable. Figure 15 compares the throughput of the three algorithms with a varying number of threads when warehouse=112 under TPC-C workload. Each algorithm has good scalability in memory mode, which shows that ART and MVCC are in sound design. Moreover, it shows that our modification of ART does not destroy scalability. However, in the checkpoint mode, the scalability is relatively poor because the I/O of logging leads to an inevitable transaction bottleneck.

5.2.6 Memory footprint. Figure 16 shows the memory footprint of different algorithms in the same configuration (warehouse=16, thread=16) running for 150 s. At the same time, all the algorithms throughput limit is to the same 30,000 per second. Note that we do not compare *ChainIndex*, as the performance of *ChainIndex* is severely degraded in the checkpoint phase. (i) *MirrorIndex* has the fastest memory growth speed because it needs an additional

MirrorIndex tree. (ii) The memory growth speed of *IACoW* is 10% more than that of *Baseline* because *IACoW* needs some additional memory, such as the indirection array. (iii) Copy on write must cause extra memory usage, and that memory is released after the checkpoint ends. Therefore, the memory of *IACoW* will expand rapidly during the checkpoint phase, and after the checkpoint, it will fall back quickly.

5.3 Summary

Based on the above evaluation, we conclude the following findings:

1. Logging does not lead to performance stall (cf. Figure 9(a)), but checkpoint does, especially for *ChainIndex* (cf. Figure 9(b)).
2. *Baseline*'s performance is the best (cf. Figure 13 and Figure 14), but its recovery time is the longest due to index rebuilding (cf. Figure 11 and Figure 12). Therefore, *Baseline* is very suitable for the workload that is not strict with recovery time.
3. Because of reading amplification (cf. Figure 13(b) and 13(c)) and garbage collection (cf. Figure 13(d)), *ChainIndex* has poor performance. The performance with the write-heavy workload is relatively better (cf. Figure 13(a)).
4. Due to write amplification, *MirrorIndex* is not suitable for the write-heavy workload (cf. Figure 13(b)).
5. *IACoW*'s performance is the most stable under different workloads (cf. Figure 13 and Figure 14). However, *IACoW* leads to a specific read amplification due to the reason of "logical pointer". Thus, the performance with read-heavy workloads is not as good as *Baseline* (cf. Figure 13).
6. Both the checkpoint time of *ChainIndex* and *MirrorIndex* is very long (cf. Figure 10(a)), and it does not support frequent checkpoints, especially, the recovery time will be very long in the case of large data sets (cf. Figure 11(a)). On the other hand, the checkpoint speed of *IACoW* is fast (cf. Figure 10(b)), it supports frequent checkpoint (cf. Figure 9(b)) and instant recovery (cf. Figure 11 and Figure 12).
7. IACoW does not cause too much extra memory footprint (Figure 16).

6 RELATED WORK

This section reviews some designs for checkpoints and indexes.

Checkpoint for IMDBs. A large number of algorithms have been proposed to take tuple checkpoint for single-version storage systems, *e.g.*, COU [5, 21, 36], fork [10], Zigzag [3], Pingpong [3], CALC [34], *etc.* Moreover, Leon Lee (*a.k.a*, Liang Li) also comprehensively studied those checkpoint performance in [19, 20]. On the other hand, for multi-version storage systems, we can get the snapshot directly with the Snapshot Isolation semantics. However, taking index checkpoint is quite complex since the tree structure is much more complex than the tuple layout, and the index is not stored in multi-versions. Therefore, as discussed in subsection 2.2, some systems always do not take index checkpoints and are recovered through rebuilding, *e.g.*, VoltDB [27], SiloR [49], PACMAN [44], WBL [1], Zen [23], *etc.* Unfortunately, wholly rebuilding index means that more time to recovery, to pursue instant recovery, especially in cloud-native systems, we think index checkpoint is indispensable.

Checkpoint for DRDBs. The checkpoint and recovery of DRDBs are very different from IMDBs. DRDBs always utilize buffer managers to accelerate page I/O, thus checkpoints are taken by flushing dirty pages within the buffer pool, which does not distinguish between indexes and tuples [7]. Since both tuples and indexes are stored as pages and the physiological log records page modifications [29], the recovery of indexes is similar to tuples. However, due to index page splits, recovering index requires additional "compensate" system transactions [6, 24]. PostgreSQL [9] and LeanStore [8] are two good examples. In summary, the index rebuilding issue does not arise in DRDBs. It is a new challenge in IMDBs.

Indexes for IMDBs. The tree-based index widely used in IMDBs can be divided into three categories. (i) B+ tree: CSB+ trees [33], and Bw-Trees [17, 42] store keys horizontally side-by-side in the leaf nodes and have good range query performance. (ii) Trie: ART [14–16] and HOT [2] is faster than B+trees on modern hardware. Both of them are memory efficient. (iii) Hybrid tree: Masstree [28] combines the B+tree and Trie designs in a single data structure. [42] comprehensively compares those trees, and ART is always the winner for most workloads. Thus we choose ART in HiEngine.

7 APPLICABILITY TO OTHER SYSTEMS

Our designs can be applied to other indexes. Despite the fact that we created and validated our work on HiEngine using adaptive radix trees, our index checkpoint designs are still applicable to other indexes, such as B+ tree. *ChainIndex* and *MirrorIndex* can be directly used within B+ tree, however for *IACoW*, some adaptation work for NodeManager (described in Listing 4) is required due to the Structure Merge Operations (SMO) of the B+ tree.

8 CONCLUSION

As far as we know, this paper is the first work comprehensively discussing index checkpoint technology for IMDBs. We observe that once the state-of-art techniques have been applied, index rebuilding becomes the new bottleneck of recovery processing in IMDBs. In this paper, we show that taking index checkpoints can significantly reduce the recovery time of index rebuilding. We combine two techniques to ensure the data correctness for transaction inconsistent index checkpoints. We design and implement three wait-free index checkpoint algorithms within a production IMDB and analyze their pros and cons in detail. End-to-end experiments show that index checkpoints can significantly shorten the recovery time, especially for the planned shutdown scenario. Specifically, *IACoW*'s performance is the most stable under various workloads. Thus we strongly recommend that *IACoW* should be integrated into IMDBs if the recovery time is a crucial product metric.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and suggestions.

REFERENCES

[1] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *Proc. VLDB Endow.* 10 (2016), 337–348.

[2] Robert Binna, Eva Zangerle, M. Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD.*

[3] Tuan Cao, M. V. Salles, B. Sowell, Yao Yue, A. Demers, J. Gehrke, and Walker M. White. 2011. Fast checkpoint recovery algorithms for frequently consistent applications. In *SIGMOD*.

[4] D. DeWitt, R. Katz, F. Olken, Leonard D. Shapiro, M. Stonebraker, and D. Wood. 1984. Implementation techniques for main memory database systems. In *SIGMOD*.

[5] J. Gehrke and Tuan Cao. 2013. Fault tolerance for main-memory applications in the cloud.

[6] Goetz Graefe. 2012. A survey of B-tree logging and recovery techniques. *ACM Transactions on Database Systems (TODS)* 37 (2012), 1 – 35.

[7] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Comput. Surv.* 15 (1983), 287–317.

[8] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD*.

[9] Hironobu. [n.d.]. The internals of postgresql for database administrators and system developers. http://www.interdb.jp/pg/

[10] A. Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. 195–206.

[11] Kangnyeon Kim, Tianzheng Wang, R. Johnson, and I. Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD*.

[12] P. Larson, Spyros Blanas, C. Diaconu, C. Freedman, J. M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5 (2011), 298–309.

[13] Viktor Leis, Michael Haubenschild, A. Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. *ICDE*, 185–196.

[14] Viktor Leis, Michael Haubenschild, and T. Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE TKDE* 42 (2019), 73–84.

[15] Viktor Leis, A. Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. *ICDE*, 38–49.

[16] Viktor Leis, Florian Scheibner, A. Kemper, and T. Neumann. 2016. The ART of practical synchronization. In *DaMoN '16*.

[17] Justin J. Levandoski, D. Lomet, and S. Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. *ICDE*, 302–313.

[18] Justin J. Levandoski, D. Lomet, and S. Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *Proc. VLDB Endow.* 6 (2013), 877–888.

[19] L. Li, Guoren Wang, Gang Wu, and Ye Yuan. 2018. Consistent Snapshot Algorithms for In-Memory Database Systems: Experiments and Analysis. *ICDE*, 1284–1287.

[20] L. Li, Guoren Wang, Gang Wu, Ye Yuan, Lei Chen, and Xiang Lian. 2021. A Comparative Study of Consistent Snapshot Algorithms for Main-Memory Database Systems. *IEEE TKDE* 33 (2021), 316–330.

[21] Antti-Pekka Liedes and A. Wolski. 2006. SIREN: A Memory-Conserving, Snapshot-Consistent Checkpoint Algorithm for in-Memory Databases. *ICDE*, 99–99.

[22] Hyeontaek Lim, M. Kaminsky, and D. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD*.

[23] Gangcai Liu, Leying Chen, and Shimin Chen. 2021. Zen: a High-Throughput Log-Free OLTP Engine for Non-Volatile Main Memory. *Proc. VLDB Endow.* 14 (2021), 835–848.

[24] David B. Lomet and Betty Salzberg. 1997. Concurrency and recovery for index trees. *The VLDB Journal* 6 (1997), 224–240.

[25] Yunus Ma, Siphrey Xie, Henry Zhong, Leon Lee, and King Lv. 2022. HiEngine:How to Architect a Cloud-Native Memory-Optimized Database Engine.. In *SIGMOD*.

[26] Arlino Magalhães, Angelo Brayner, José Maria S. Monteiro, and Gustavo Moraes. 2021. Indexed Log File: Towards Main Memory Database Instant Recovery. In *EDBT*.

[27] Nirmesh Malviya, Ariel Weisberg, S. Madden, and M. Stonebraker. 2014. Rethinking main memory OLTP recovery. *ICDE*, 604–615.

[28] Yandong Mao, E. Kohler, and R. Morris. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys '12*.

[29] C. Mohan, Donald J. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.* 17 (1992), 94–162.

[30] Thomas Neumann, Tobias Mühlbauer, and A. Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*.

[31] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, T. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, A. Tomasic, S. Toor, D. V. Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR*.

[32] PMDK. [n.d.]. using-persistent-memory-devices-with-the-linux-device-mapper. https://pmem.io/2018/05/15/using_persistent_memory_devices_with_the_linux_device_mapper.html

[33] J. Rao and K. A. Ross. 2000. Making B+- trees cache conscious in main memory. In *SIGMOD*.

[34] Kun Ren, Thaddeus Diamond, D. Abadi, and Alexander Thomson. 2016. Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems. In *SIGMOD*.

[35] K. Salem and H. Garcia-Molina. 1990. System M: A Transaction Processing Testbed for Memory Resident Data. *IEEE TKDE* 2 (1990), 161–172.

[36] M. V. Salles, Tuan Cao, B. Sowell, A. Demers, J. Gehrke, Christoph E. Koch, and Walker M. White. 2009. An Evaluation of Checkpoint Recovery for Massively Multiplayer Online Games. *Proc. VLDB Endow.* 2 (2009), 1258–1269.

[37] Caetano Sauer. 2019. Modern techniques for transaction-oriented database recovery1.

[38] Caetano Sauer, G. Graefe, and T. Härder. 2017. Instant Restore After a Media Failure. In *ADBIS*.

[39] Caetano Sauer, G. Graefe, and T. Härder. 2018. FineLine: log-structured transactional storage and recovery. *Proc. VLDB Endow.* 11 (2018), 2249–2262.

[40] Stephen Tu, Wenting Zheng, E. Kohler, B. Liskov, and S. Madden. 2013. Speedy transactions in multicore in-memory databases. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*.

[41] Alexandre Verbitski, Anurag Gupta, D. Saha, Murali Brahmadesam, K. Gupta, Raman Mittal, S. Krishnamurthy, Sandor Maurice, T. Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*.

[42] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, M. Kaminsky, and D. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*.

[43] Yingjun Wu, Joy Arulraj, Jiexi Lin, R. Xian, and A. Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10 (2017), 781–792.

[44] Yingjun Wu, Wentian Guo, Chee-Yong Chan, and Kian-Lee Tan. 2017. Fast Failure Recovery for Main-Memory DBMSs on Multicores. In *SIGMOD* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 267–281. https://doi.org/10.1145/3035918.3064011

[45] Yu Xia, Xiangyao Yu, A. Pavlo, and S. Devadas. 2020. Taurus: Lightweight Parallel Logging for In-Memory Database Management Systems (Extended Version). *ArXiv* abs/2010.06760 (2020).

[46] Chang Yao, D. Agrawal, G. Chen, B. Ooi, and Sai Wu. 2016. Adaptive Logging: Optimizing Logging and Recovery Costs in Distributed In-memory Databases. In *SIGMOD*.

[47] Hao Zhang, Gang Chen, B. Ooi, K. Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE TKDE* 27 (2015), 1920–1948.

[48] L. Zhang, Matthew Butrovich, Tianyu Li, Yash Nannapanei, A. Pavlo, John Rollinson, Huanchen Zhang, Ambarish Balakumar, Daniel Biales, Ziqi Dong, Emmanuel Eppinger, J. González, Wan Shen Lim, Jianqiao Liu, Prashanth Menon, Soumil Mukherjee, Tanuj Nayak, Amadou Ngom, Jeff Niu, Deepayan Patra, Poojita Raj, Stephanie Wang, Wuwen Wang, Yao Yu, and W. Zhang. 2020. Everything is a Transaction: Unifying Logical Concurrency Control and Physical Data Structure Maintenance in Database Management Systems.

[49] Wenting Zheng, Stephen Tu, E. Kohler, and B. Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*.