



Swan: Hybrid MVCC Management for Efficient Transaction Processing in LSM-Tree-Based Key-Value Stores

Yang Guo*

The Chinese University of Hong Kong
yangguo@cse.cuhk.edu.hk

Jin Xue*

The Chinese University of Hong Kong
csjinxue@outlook.com

Zili Shao

The Chinese University of Hong Kong
shao@cse.cuhk.edu.hk

ABSTRACT

Distributed NewSQL databases build SQL query engines on top of LSM-tree-based key-value stores to achieve fast and scalable online transaction processing. This requires efficient transaction support in the underlying storage engine, typically implemented using multi-version concurrency control (MVCC). However, existing in-memory MVCC schemes struggle with large write transactions due to high memory consumption and commit overhead, which can degrade insertion performance or even cause system crashes. Out-of-memory MVCC schemes alleviate these issues but suffer from expensive visibility checks under high concurrency, leading to performance degradation. In this paper, we propose a hybrid MVCC scheme that combines the fast visibility checks of in-memory MVCC with the low commit latency and memory usage of out-of-memory MVCC, enabling efficient transaction management. To integrate this design with LSM-tree-based key-value stores, we introduce several memory management techniques, including a transaction-aware data separation strategy, a stash region for uncommitted data, and a concurrent memtable flushing mechanism. We implement a prototype key-value store and extend YCSB to evaluate our design under various workloads. Experimental results demonstrate that our hybrid MVCC scheme combines the strengths of both approaches and delivers improved performance.

PVLDB Reference Format:

Yang Guo, Jin Xue, and Zili Shao. Swan: Hybrid MVCC Management for Efficient Transaction Processing in LSM-Tree-Based Key-Value Stores. PVLDB, 19(9): 1977 - 1990, 2026.
doi:10.14778/3819518.3819528

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/LAccordeur/swan>.

1 INTRODUCTION

In recent years, to address the growing variety of workloads in large-scale data applications [7, 11, 18, 45], NewSQL databases [3, 7, 13, 20, 32, 37] have emerged, integrating ACID (atomicity, consistency, isolation, and durability) transaction features into LSM-tree-based key-value stores. This integration enables NewSQL databases to combine the high performance of LSM-tree-based key-value

stores with the ACID guarantees necessary for online transaction processing. As a result, providing efficient transaction support in LSM-tree-based key-value stores has become increasingly critical to implement efficient NewSQL databases.

To support transaction processing, multi-version concurrency control (MVCC) is commonly used in existing databases [5, 24, 33, 36, 49, 50]. In an MVCC-based system, each write operation to a key-value pair creates a new version, which is associated with a timestamp indicating the time of its commitment [1]. A transaction can read any version committed before it starts. Two key operations in this process are commit and visibility check. For example, one responsibility of the commit operation is to update the timestamp of each affected version to the transaction's commit timestamp. The visibility check operation, on the other hand, determines whether a version is visible to a transaction by comparing its timestamp with the transaction's start timestamp. An efficient MVCC scheme should optimize performance for both of these operations.

Although some existing key-value stores [20, 30, 41, 56], such as RocksDB [41], adopt MVCC schemes to manage transactions, the in-memory MVCC scheme [14, 28, 36] commonly used in these systems struggles to efficiently handle large write transactions [42]. Specifically, in in-memory MVCC, all uncommitted data must be buffered in memory to avoid disk access during the commit phase, leading to substantial memory consumption. For instance, RocksDB stores all uncommitted updates in a transaction-local write batch [41], which can cause out-of-memory errors when processing large write transactions, as shown in Subsection 2.3. This problem is further exacerbated in memory-constrained environments. In addition, committing large write transactions incurs high overhead due to the need to update timestamps for numerous versions. To reduce memory and commit overhead, some systems adopt an out-of-memory MVCC scheme [34, 39, 43]. This approach avoids updating timestamps during transaction commits and allows uncommitted data to be written to disk. However, without commit timestamps, visibility checks require additional lookups in internal data structures (such as a shared map that tracks transaction states) to determine whether a transaction has committed. This introduces high overhead, especially in concurrent environments, due to the extra work during each visibility check and the increased contention on shared internal structures involved in the process.

To efficiently handle large write transactions and provide high-performance transaction processing in concurrent environments, we propose a hybrid MVCC scheme, Swan. It adopts the out-of-memory MVCC approach for large write transactions and the in-memory MVCC approach for normal (typically small) transactions. Compared to pure in-memory MVCC schemes, Swan reduces commit latency and memory usage for large write transactions. Compared to pure out-of-memory MVCC schemes, Swan significantly

*Both authors contributed equally to this research.

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. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819528

lowers visibility check overhead and reduces contention in concurrent environments, as the number of concurrent transactions using the out-of-memory scheme is minimized. This hybrid scheme is beneficial for workloads that mix large write transactions with normal transactions, such as those found in financial platforms [16].

However, enabling efficient transaction processing with the hybrid MVCC scheme in LSM-tree-based key-value stores presents two key challenges. First, unlike in-memory or disk-based databases, LSM-tree-based systems involve both memory and disk components. A major challenge is how to integrate two distinct MVCC schemes to ensure fast commits and visibility checks across both components. Second, LSM-tree-based systems perform additional operations, such as memtable flushing and compaction. Coordinating these operations with hybrid MVCC management is essential to achieve seamless integration and maintain high performance.

To address the first challenge, the key lies in effectively managing versions and their timestamps across both the memory and disk components to ensure efficient visibility checks and commit operations. We observe that LSM-tree-based key-value stores are inherently multi-version: different versions of a key may exist in the memtable and multiple SSTables, and each location has distinct characteristics. Based on this, we propose different strategies for managing the timestamps of versions at these various locations. First, for versions in the active memtable, which are in-memory and mutable, we adopt a hybrid approach to manage the timestamps of versions created by different types of transactions. Specifically, for versions created by small normal transactions, we use an in-memory MVCC scheme to handle timestamps, enabling fast visibility checks. For versions created by large write transactions, we rely on an out-of-memory MVCC scheme to handle timestamps for faster commits. Second, for versions in immutable memtables or in level 0 (L_0) of the LSM-tree, which may have overlapping keys across different files, we maintain a per-file timestamp to distinguish the freshness of versions across files. Third, for versions in the lower levels of the disk component, timestamps are discarded, since version freshness can be determined directly from their level in the LSM-tree.

To address the second challenge, the key lies in effectively managing the active memtable. Existing LSM-tree-based key-value stores flush a full active memtable either entirely to disk or based on key access frequency [4, 32], without considering transaction states. We propose a transaction-aware memory management approach with three key techniques. First, we introduce a transaction-aware data separation strategy that distinguishes between committed and uncommitted data based on the transaction states. With this strategy, only committed versions and uncommitted versions from large write transactions are flushed to disk. Second, we design a *stash region* on disk to temporarily store files containing uncommitted versions from large write transactions. Once the corresponding transaction commits, these files are promoted to L_0 of the LSM-tree. This stash region isolates uncommitted files from committed ones, enabling lightweight integration with the existing compaction process and efficient rollback by simply removing the associated files. Third, we propose a concurrent flushing mechanism to accelerate the flushing process of memtables under high write pressure.

We implement Swan in a prototype key-value store and extend the YCSB benchmark [12] to evaluate its transaction processing performance under various workloads. Experimental results show

that Swan combines the strengths of both in-memory and out-of-memory MVCC schemes. By incorporating the out-of-memory MVCC scheme, Swan handles large write transactions with reduced memory usage and lower commit overhead, improving insertion throughput by up to 19% compared to the in-memory MVCC scheme. At the same time, Swan delivers comparable performance to the in-memory MVCC scheme on most transactional workloads under concurrent execution. The implementation is open source [52]. In summary, we make the following contributions:

- We design a hybrid MVCC scheme that enables fast visibility checks and commits while reducing memory usage for large write transactions, supporting efficient transaction processing in concurrent environments (§3).
- We propose a transaction-aware memory optimization and a stash region on disk to store uncommitted data. These techniques enable lightweight integration of the hybrid MVCC scheme into LSM-trees (§4).
- We demonstrate how to integrate the hybrid MVCC scheme into LSM-tree-based key-value stores, covering data flushing and compaction, in a lightweight and efficient way. Evaluation on YCSB benchmarks highlights the effectiveness and efficiency of our approach (§6).

2 BACKGROUND AND MOTIVATION

2.1 LSM-Tree-Based Key-Value Stores

2.1.1 LSM-Tree Architecture. The typical architecture of an LSM-tree-based key-value store [10, 32, 51] consists of three main components: the memory component, the disk component, and the on-disk write-ahead log (WAL).

The *memory component* is an in-memory data structure that maps keys to values and consists of an active memtable and possibly several immutable memtables. The active memtable acts as a write buffer, holding recent updates. When it reaches a size limit, it becomes immutable, and a new active memtable is created. The immutable memtable is then flushed to the first level (L_0) of the disk component as a new SSTable. Data in the memory component becomes persistent only after being flushed to disk. To avoid data loss, all updates to the memory component are first appended to the on-disk *write-ahead log*, which is replayed after a crash to restore the memory component.

The *disk component* is organized into multiple levels, from L_0 to L_{n-1} . Each level contains multiple sorted files, known as SSTables. The first level, L_0 , is special because its SSTables are created by flushing the memtable and therefore may have overlapping key ranges. In contrast, SSTables in the other levels (L_i , where $i \neq 0$) are generated during the compaction process, in which the LSM-tree periodically sort-merges SSTables from level i with overlapping SSTables from level $i+1$. There are various compaction strategies in LSM-tree-based key-value stores, such as leveled compaction and size-tiered compaction [44, 47]. In this paper, we assume leveled compaction is used; however, the proposed method can also be adapted to other strategies like size-tiered compaction.

2.1.2 Operations. The two basic operations in an LSM-tree-based key-value store are $Get(txn, k)$ and $Put(txn, k, v)$, both executed within a transaction context txn . $Put(txn, k, v)$ inserts the pair $(k,$

v) into the memory component and appends a record to the WAL. $Get(txn, k)$ retrieves the latest committed value of key k as of the transaction’s start time. It first searches the active memtable, then the immutable memtables. If not found, it scans all files in L_0 , since their key ranges may overlap. If still not found, it continues from L_1 to L_{n-1} , checking only one file per level because files in these lower levels have disjoint key ranges.

2.2 Multi-Version Concurrency Control

Multi-version concurrency control (MVCC) is widely used in modern databases to enable concurrent access. Instead of updating data in place, MVCC creates a new version for each write. This allows transactions to read previously committed versions while other transactions update the same key, with conflicts occurring only when multiple transactions attempt to update it concurrently. Outdated versions are later removed through garbage collection. An important aspect of MVCC is efficiently determining which versions are visible to a given transaction. In practice, databases use two main MVCC variants: the in-memory MVCC scheme and the out-of-memory MVCC scheme, which differ in how they handle visibility checks and commit operations [1].

2.2.1 In-Memory MVCC. In-memory MVCC schemes are commonly used in main-memory databases such as HANA [17], Hekaton [14], and Hyper [36]. In this scheme, each transaction is assigned a start timestamp from a global counter, and each version is associated with the start timestamp of the transaction that created it. Upon commit, the transaction receives a commit timestamp from the same counter, and all versions in its write set are updated with this new timestamp. Note that some in-memory systems (e.g., HANA) avoid direct per-version timestamp updates. Instead, versions created by the same transaction may indirectly reference a shared transaction or group-commit context, through which the commit timestamp is propagated asynchronously to all related versions [17]. A transaction determines version visibility using the version’s timestamp, either stored directly in the version or obtained indirectly via the shared transaction context. A start timestamp indicates an uncommitted version, while a commit timestamp is compared with the transaction’s start timestamp to determine visibility. In-memory MVCC offers fast reads because each visibility check is lightweight, generally involving only a timestamp comparison and, when indirection is employed, a pointer dereference. However, commits can incur higher overhead for large write transactions due to the need to coordinate commit metadata across a large write set, particularly when the commit process requires re-stamping the write set with the commit timestamp.

2.2.2 Out-of-Memory MVCC. Out-of-memory MVCC schemes are widely used in disk-based databases such as PostgreSQL [39] and MySQL InnoDB [34]. Each transaction is assigned a transaction ID (TXID) from a global counter, which is stored in the versions it creates. The system maintains a central structure to track active TXIDs. When a transaction starts, it creates a snapshot. A snapshot consists of the set of active TXIDs and an upper bound on TXIDs (i.e., the maximum or next TXID at the time the snapshot is created). Thus, a snapshot may contain up to T TXIDs, where T is the number of concurrent transactions in the system. During reads, the system

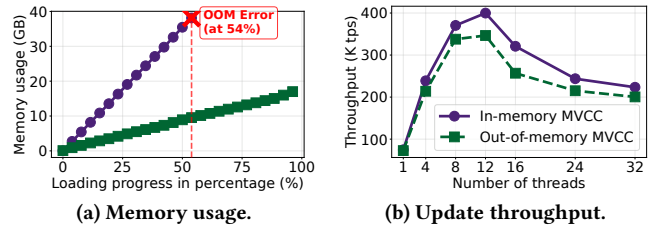


Figure 1: Evaluation of in-memory MVCC and out-of-memory MVCC on RocksDB.

determines whether a version with TXID x is visible by checking the following two conditions: (i) x is not in the snapshot’s active TXID set, indicating that the version was not created by a concurrent transaction; and (ii) x is smaller than the snapshot’s upper bound, indicating that the version was not created by a future transaction. This approach enables fast commits because it avoids re-stamping versions at commit time, which is especially useful in disk-based systems where accessing flushed versions may require reading them back into memory for re-stamping.

However, visibility checks in out-of-memory MVCC become more complex, potentially requiring up to T timestamp comparisons and causing contention on shared data structures, such as the global active transaction set, which can degrade performance, particularly under high concurrency. Although auxiliary data structures can reduce some timestamp comparisons (e.g., materializing the active set in a transaction’s snapshot as a thread-local hash set for condition (i)), the snapshot itself must still be constructed, typically at transaction start. Constructing a correct snapshot requires consulting the global active transaction set together with an upper bound on TXIDs. As the number of concurrent transactions increases, snapshot construction incurs $O(T)$ work and repeatedly accesses the shared structure that is updated on every transaction start and commit. Consequently, even when per-check comparisons are reduced, snapshot construction on shared structures remains costly and introduces contention under high concurrency.

2.3 Motivation

In LSM-tree-based key-value stores such as RocksDB, in-memory MVCC is used to support transactions. To avoid disk access during a commit, all uncommitted versions are kept in memory. In RocksDB, these versions are stored in a transaction-local write batch and merged into the memtable only at commit. However, this approach can consume a large amount of memory for large write transactions, sometimes causing out-of-memory errors and system crashes, as shown in Figure 1a. Moreover, as shown in Subsubsection 6.2.1, the commit latency increases noticeably with larger transactions. For example, it rises from 26.8 milliseconds to 43.3 milliseconds when committing a write transaction with 50,000 operations, mainly due to overhead such as re-stamping the write set.

To address the memory pressure and commit latency issues associated with large write transactions, RocksDB implements Write-Unprepared Transactions [43], which follow the out-of-memory MVCC scheme. This approach allows uncommitted versions to be written directly into the memtable, which significantly reduces memory usage, as shown in Figure 1a. To perform visibility checks on versions stored in the memtable, additional data structures are

Table 1: Comparison among different MVCC schemes.

Scheme	Txn. size	Commit latency	Visibility check
In-memory MVCC [1]	Small	$\Theta(\text{write set})$	$\Theta(1)$
	Large	$\Theta(\text{write set})$	$\Theta(1)$
Out-of-memory MVCC [1]	Small	$\Theta(1)$	$\Omega(1), O(T)$
	Large	$\Theta(1)$	$\Omega(1), O(T)$
Hybrid MVCC	Small	$\Theta(\text{write set})$	$\Omega(1), O(T')$
	Large	$\Theta(1)$	$\Omega(1), O(T')$

* T is the number of all concurrent transactions, while T' is the number of concurrent large transactions.

maintained to track the commit status of the transactions that created these versions [42, 43]. However, this approach increases visibility check overhead, requiring more data structure lookups and timestamp comparisons. Moreover, relying on these internal data structures for visibility checks can lead to contention in concurrent environments, which degrades performance. As shown in Figure 1b, in a concurrent setting, the update throughput of a transactional workload on RocksDB drops by up to 20%, and similar degradation is also observed on some other workloads, such as write-read workloads. This performance decline of the out-of-memory MVCC scheme in concurrent environments is further validated by our prototype system, as shown in Subsection 6.3.

We also observe that workloads combining large write transactions with normal read/write transactions are common in practice. For instance, a financial platform may handle many read- or update-heavy transactions from concurrent clients in the morning, and large write transactions, such as bulk imports of up to 2.5 million entries per commit, in the evening [16]. As discussed above, a single MVCC scheme cannot efficiently handle such mixed workloads.

Based on these observations, we propose a hybrid MVCC scheme that combines the strengths of in-memory and out-of-memory approaches: out-of-memory MVCC is used for large write transactions, while in-memory MVCC handles small normal transactions. Compared to pure in-memory MVCC, the hybrid approach can manage large write transactions with lower memory usage by allowing uncommitted data to be flushed to disk. It also reduces the commit latency for large transactions, as shown in Table 1. Compared to pure out-of-memory MVCC, it lowers visibility check overhead and contention under concurrency. For example, for versions created by the in-memory MVCC scheme, only a single timestamp comparison is needed. For versions created by the out-of-memory MVCC scheme, the number of timestamp comparisons is significantly reduced by limiting the number of concurrent transactions that use the out-of-memory MVCC scheme. However, integrating the hybrid MVCC scheme into LSM-tree-based key-value stores in a way that seamlessly cooperates with both memory and disk components, as well as with internal operations such as data flushing and compaction, in an efficient and lightweight manner, is non-trivial. Overcoming this challenge is the focus of our paper.

3 HYBRID MVCC MANAGEMENT

In this section, we present Swan, a hybrid MVCC scheme. We first introduce the required metadata in Subsection 3.1, and then describe hybrid transaction processing in the memory component in Subsection 3.2. We defer the discussion of transaction processing for disk-resident versions to Section 4.

3.1 Transaction Metadata

We support two types of transactions in the system: *normal* transactions and *fast-commit* transactions. Normal transactions, which include read-only operations and small write transactions, are managed using an in-memory MVCC scheme to enable lightweight visibility checks under high concurrency. Fast-commit transactions, typically large transactions involving many writes or updates, are managed using an out-of-memory MVCC scheme to reduce commit latency and memory usage. Versions created by normal transactions are called normal versions (or in-memory MVCC versions), while those from fast-commit transactions are called fast-commit versions (or out-of-memory MVCC versions). The user specifies the transaction type when starting the transaction. Automatically determining the transaction type at runtime is left for future work.

To distinguish between the two transaction types, we use a timestamp-based MVCC scheme. Each timestamp contains a 62-bit integer value and a 2-bit type tag. The value is generated by a global monotonic counter, and the type tag specifies how the value should be interpreted. In this paper, we represent a timestamp as a tuple (**type**, **value**). The three possible types are described below.

- **Uncommitted (U)**: The value part represents the start time of an in-progress normal transaction.
- **Commit timestamp (C)**: The value part represents the commit time of a normal transaction.
- **TXID (T)**: The value part represents the TXID of a fast-commit transaction.

Note that each transaction is assigned a start timestamp upon creation, which also serves as its transaction ID (TXID). Next, we introduce the major metadata used to manage transactions.

Write Set. Each *normal transaction* maintains a *write set*, which holds references to all uncommitted versions it creates in the active memtable. When a new version is generated, it is initially marked with a temporary timestamp (i.e., (**U**, *start timestamp*)) to prevent other transactions from reading uncommitted data. Upon commit, a commit timestamp is assigned, and all versions in the write set are re-stamped with this final timestamp (i.e., (**C**, *commit timestamp*)), making them visible to subsequent transactions. In contrast, fast-commit transactions do not maintain this metadata.

Snapshot. A normal or fast-commit transaction can have a *snapshot*, which consists of the TXIDs of all active fast-commit transactions and an upper bound on TXIDs (e.g., the maximum TXID at the time the snapshot is created). The snapshot determines the visibility of versions created by fast-commit transactions. It is generated at the start of a transaction using an internal data structure that records all active fast-commit transactions. With the proposed hybrid MVCC scheme, fewer fast-commit transactions are tracked in this structure, reducing visibility check overhead and lowering contention from thread access in concurrent environments.

Flush Set. When the uncommitted versions of a fast-commit transaction are flushed to the disk, we record the keys of versions for this fast-commit transaction. This *flush set* can later be utilized when performing a write-write conflict check. When a fast-commit transaction commits, it will remove the associated flush set. For extremely large write transactions, *flush set* may lead to increased memory pressure. This can be addressed by spilling *flush set* to disk once it exceeds a memory budget or by using compact summaries.

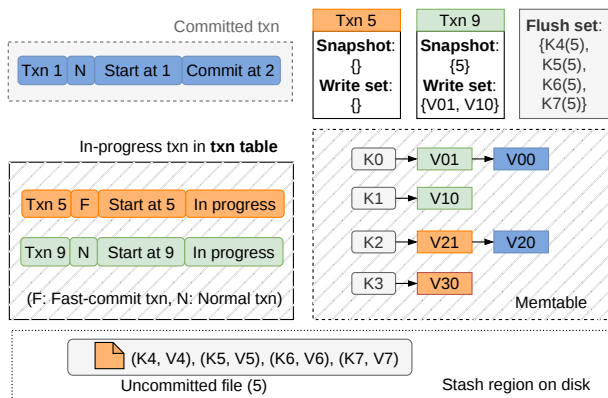


Figure 2: The hybrid MVCC example. In our memtable implementation, each key maintains a version chain where versions are ordered by their insertion time. The color of each version indicates the transaction that created it.

Transaction Table. We maintain a central in-memory *transaction table* to track the TXIDs of all in-progress transactions. Entries for both normal and fast-commit transactions are removed upon commit. For fast-commit transactions, abort outcomes are kept in the table for visibility checks in potential cases, as shown in Algorithm 1. The transaction table does not need to be flushed to disk, since it is only used for visibility checks on versions in the active memtable. After a crash, it can be reconstructed from the commit log records in the WAL.

EXAMPLE 1. With the above metadata, the two MVCC schemes are seamlessly integrated. As illustrated in Figure 2, there is one committed transaction (Txn 1) that committed at timestamp 2, and two in-progress transactions: one fast-commit transaction (Txn 5) that started at timestamp 5, and one normal transaction (Txn 9) that started at timestamp 9. Txn 1 created versions V00 and V20, both of which are committed. Txn 5 created versions V21 and V30, which are still uncommitted. Txn 9 created versions V01 and V10, which are also uncommitted. In this example, Txn 5 has flushed some data into the stash region on disk. As a result, the flush set records the keys of the flushed versions along with the corresponding timestamp 5 (i.e., the TXID of Txn 5). For Txn 9, its snapshot includes Txn 5, as Txn 5 was the only active fast-commit transaction when Txn 9 began. For simplicity, the example figure omits the upper bound on TXIDs in the snapshot. Txn 9’s write set contains references to its own uncommitted versions.

3.2 Transaction Processing in the Memory Component

In our design, instead of buffering uncommitted data in a private space of a transaction until commit time, we insert uncommitted data directly into the active memtable. This avoids the data copy overhead during commit. As shown in Figure 2, each key in the memtable maintains a version chain, where versions are ordered by their insertion time. Each version contains a *timestamp*, which is used by reader transactions to determine visibility. The specific organization of keys in the memtable can be a hash map or a skip list, or others, depending on the implementation.

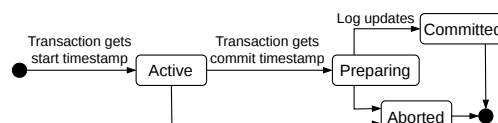


Figure 3: State transitions for a transaction.

In the following, we introduce the transaction processing flow and explain how visibility checks and commit operations are implemented efficiently. Figure 3 shows the life cycle of a transaction [26].

3.2.1 Begin. When a transaction begins, the global monotonic counter is incremented to generate a new start timestamp. This timestamp is used to determine which versions are visible to the transaction during reads. The client can specify whether to initiate a normal transaction, which follows the in-memory MVCC scheme, or a fast-commit transaction, which follows the out-of-memory MVCC scheme. Once the transaction context is established, the client can begin performing operations on the active memtable.

3.2.2 Put. In the active memtable, a *put* operation either inserts an initial key-value pair or overwrites the existing value for a given key. After locating or creating an entry for the key k , a new version is created and added to its version chain, as illustrated in Figure 2. The new version stores the start timestamp (or TXID), tagged as UNCOMMITTED for normal transactions or TXID for fast-commit transactions. Once the new version is prepared, the current value of the version pointer (i.e., points to the current head of the version chain) is first loaded, and the next pointer of the new version is set to the current head of the version chain. Then, a compare-and-swap (CAS) operation is used to atomically install the new version into the version pointer, ensuring that in concurrent updates, only the first thread succeeds. Additionally, if no entry exists for the key, the system checks the in-memory flush set. If the key is present, the transaction is aborted to avoid updating an uncommitted version created by another transaction.

In our hybrid MVCC design, even though versions may originate from different MVCC schemes and timestamps have different meanings, such as normal versions carrying a commit timestamp and fast-commit versions retaining only a start timestamp even after commit, we can still determine their correct ordering. For example, consider a fast-commit version with TXID x (its start timestamp), and a normal version for the same key with a commit timestamp of $x+N$. Even though the fast-commit version does not explicitly store its commit timestamp, it must have committed before the normal version. This is guaranteed by the conflict detection mechanism, which enforces a consistent ordering among conflicting writes.

3.2.3 Get. In an MVCC scheme, such as for snapshot isolation, a transaction can only read versions of a key created before it began. To access a visible version of a key, transaction T traverses the version chain for that key and stops at the first version v that passes the visibility check (i.e., the latest committed version created before T started), as illustrated in Algorithm 1. Specifically, we perform in-memory or out-of-memory visibility checks based on the type tag of the version’s timestamp. If the type is UNCOMMITTED, the version is still uncommitted and is only visible to the transaction that created it (line 4). If the type is COMMIT, the version is visible if it was committed before the current transaction started (line 7). Otherwise,

Algorithm 1: Visibility Check for Memtable Versions.

Input: Current transaction txn , $version$ to be checked

```
1 function VisibilityCheck( $txn$ ,  $version$ ):
2    $ts \leftarrow version.timestamp$ ;
3   if  $ts.type == UNCOMMITTED$  then
4     if  $ts.value == txn.start$  then
5       return VISIBLE;
6   else if  $ts.type == COMMIT$  then
7     if  $ts.value < txn.start$  then
8       return VISIBLE;
9   else
10    return VisibilityCheckSlow( $txn$ ,  $version$ );
11 function VisibilityCheckSlow( $txn$ ,  $version$ ):
12    $ts \leftarrow version.timestamp$ ;
13   if  $ts.value == txn.txid$  then
14     return VISIBLE;
15   else if  $ts.value \in txn.snapshot$  or  $ts.value >$ 
      GetMaxTXID( $txn.snapshot$ ) then
16     return INVISIBLE;
17   else
18      $state \leftarrow GetTransactionState(ts.value)$ ;
19     if  $state == ABORTED$  then
20       return INVISIBLE;
21     else
22       return VISIBLE;
```

the version was created by a fast-commit transaction, and we follow out-of-memory MVCC [1]. In this case, the version is invisible if its TXID appears in the current transaction’s snapshot or exceeds the maximum TXID in the snapshot (line 15). The first condition indicates that the version was created by a transaction active when the current transaction began, while the second indicates that it was created by a future transaction. If neither condition holds, we check the state of the creating transaction. If the transaction was aborted, the version is invisible; otherwise, it was created by a previously committed fast-commit transaction and is visible. If no visible version is found in the active memtable, the search continues in the immutable memtables and then the disk component. If the transaction has associated files in the stash region, those files are also checked, as they may contain uncommitted versions.

EXAMPLE 2. As shown in Figure 2, when Txn 9 accesses V00 (a normal version), the system compares its commit timestamp (2) with Txn 9’s start timestamp (9) and determines that the version is visible. When Txn 9 accesses a fast-commit version such as V21, it detects that the creating transaction (Txn 5) is included in its snapshot, and thus concludes that the version is not visible. With the hybrid MVCC scheme, normal and fast-commit versions can coexist in the active memtable. Normal versions use in-memory MVCC with simple timestamp comparisons, while fast-commit versions follow out-of-memory MVCC. Keeping the number of concurrent fast-commit transactions small reduces the overhead of visibility checks for these fast-commit versions.

3.2.4 Commit. Upon committing a transaction, the global monotonic counter generates a commit timestamp. When a new version is created by a normal transaction, it is initially marked with the transaction’s start timestamp as a temporary timestamp. Once the commit timestamp is obtained, we re-stamp the versions created by the normal transaction with the actual commit timestamp, making them visible to subsequent transactions. To achieve this, we traverse the versions recorded in the write set and replace their timestamps accordingly. Note that atomic commit semantics must be enforced for a transaction’s write set (e.g., via a synchronization mechanism such as a seqlock-style epoch [25]), so that readers never observe partially committed transactions. Since normal transactions typically have small write sets, this adds little latency. For fast-commit transactions, the commit process is instantaneous, as the TXIDs are directly stored in the versions created by these transactions, eliminating the need to revisit the write set. In addition, if there is associated metadata for this transaction in the flush set and stash region, we remove the metadata from the flush set and move the metadata of its uncommitted files from the stash region to L_0 management. Before a transaction is marked as committed, its WAL entries must be flushed to disk to ensure durability.

3.2.5 Abort. When a conflict is detected, one or more concurrent transactions must be aborted. For normal transactions, when an abort occurs, all of its changes are rolled back by removing its uncommitted versions from the version chains. For fast-commit transactions, given their large volume, only the aborted state is recorded in the transaction table to ensure that subsequent transactions skip the aborted versions. These versions are eventually removed during memtable flushing. Additionally, if an aborted fast-commit transaction has associated metadata or data in the flush set or stash region, these entries are removed directly.

4 TRANSACTION-AWARE MEMORY AND STORAGE MANAGEMENT

Swan allows uncommitted data to be flushed to disk to prevent memory overflow caused by large write transactions. However, this breaks the conventional assumption that disk components contain only committed data. It may also complicate LSM-tree operations, such as compactions, due to the need to consider uncommitted data. To address this, Swan introduces a *stash region* on disk to temporarily store uncommitted data files. Once the transaction commits, these files are promoted to L_0 , ensuring seamless integration with existing LSM-tree operations, including compaction. This design also enables efficient rollback.

Next, we describe how the *stash region* enables transaction-aware memtable flushing and supports compaction and rollback in Subsection 4.1, followed by version management in the disk component in Subsection 4.2 and crash recovery in Subsection 4.3.

4.1 Transaction-Aware Memtable Flushing

When a memtable becomes full, a straightforward approach is to flush it entirely to disk. However, if it contains both committed and uncommitted versions, this strategy becomes suboptimal, as it complicates operations like rollback and compaction. For instance, systems like RocksDB flush all versions together, so rolling back an

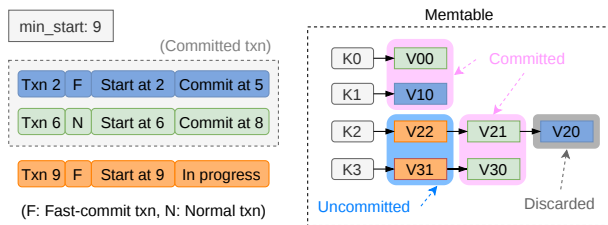


Figure 4: The collect phase of the transaction-aware flushing.

aborted transaction requires inserting extra key-value pairs to cancel previous writes [42, 43], which is costly for large transactions. Compaction also becomes slower because each version requires checking its commit status, increasing both complexity and overhead. To address this, Swan separates committed and uncommitted versions during flushing and stores uncommitted files in a stash region through the collect-and-clean procedure described next.

4.1.1 Preparation. When the active memtable becomes full, a flush is triggered, and a special timestamp `min_start` is recorded to mark the flushing job. This `min_start` is the smallest start timestamp among all transactions that have not yet committed at the time the flush begins. It is used to (1) determine the logical order of the generated SSTable and (2) distinguish committed from uncommitted versions. Once `min_start` is recorded, the system enters the collect phase, where it gathers versions of the keys. This phase produces two types of immutable tables: one containing committed versions, which is written to L_0 , and another containing uncommitted versions, which is stored in the stash region on disk. The `min_start` is also recorded as a per-file timestamp for each L_0 file to indicate its version freshness relative to other files.

4.1.2 Collect Phase. For each key, we collect one version that is visible to all in-progress transactions to form the committed immutable table. Any version older than this selected version is no longer needed and is discarded, thereby reducing the number of versions written to disk. In parallel, we also collect uncommitted versions for each key to construct the uncommitted immutable table. The version selected for each key is referred to as the tail version for the current flushing job.

In the case where `min_start` originates from a normal transaction, only the committed immutable memtable is generated. This is because any active fast-commit transaction must have started after this normal transaction, or there are no active fast-commit transactions at all. To collect versions, we examine each key in the memtable and traverse its version chain to identify the most recent version that is committed before `min_start`. For normal versions, this simply involves checking whether the commit timestamp is less than `min_start`. For fast-commit versions, we check whether the TXID (i.e., the start timestamp) is less than `min_start`. If so, and the transaction is not marked as aborted in the transaction table, it must have already committed before `min_start`, since no fast-commit transaction with an earlier TXID can still be active. All such tail versions are collected into the committed immutable table.

In the case where `min_start` originates from a fast-commit transaction, it indicates that the earliest active transaction is a fast-commit transaction. In this scenario, both a committed immutable memtable and an uncommitted immutable memtable may

be generated. When collecting versions, we handle normal and fast-commit versions differently. For normal versions, we locate the latest version that was committed before `min_start` and place it in the committed immutable memtable. For fast-commit versions, two cases must be considered. In Case 1, the TXID of the fast-commit version is smaller than `min_start`. This means the version was created by an earlier fast-commit transaction, which must have already committed or aborted. If the transaction is committed, the version is added to the committed immutable memtable (e.g., in Figure 4, $V10$ is added to a committed immutable memtable). In Case 2, the TXID of the fast-commit version is equal to `min_start`. This indicates that the version was generated by this active fast-commit transaction. Since this transaction is still in progress, the version is considered uncommitted and is placed in the uncommitted immutable memtable (e.g., in Figure 4, $V22$ and $V31$ are placed in an uncommitted immutable memtable). Additionally, if there is a previous version in the chain, it must also be collected and added to the committed immutable memtable (e.g., $V21$ and $V30$ in Figure 4). This precaution ensures that if this fast-commit transaction aborts, we do not lose the last committed version of the key. In Case 2, both files share the same per-file timestamp. How to distinguish which version is newer between the two files is discussed in Subsection 4.2.

After the collect phase, the committed immutable memtable is flushed to L_0 , while the uncommitted one is flushed to the stash region. This stash region provides several benefits. First, the rollback of an aborted fast-commit transaction is simple, as its files can be directly removed from the stash region. Second, committing a fast-commit transaction only requires moving the file metadata from the stash region to L_0 , which is lightweight. Third, uncommitted files are isolated from regular L_0 files, so the existing compaction logic can work unchanged. To support these benefits, when the uncommitted immutable memtable is flushed to the stash region, we record which files are stored there and associate each uncommitted file with its timestamp and key set in the flush set. This in-memory metadata is used to perform write-write conflict checks and reduce unnecessary disk access during reads. When a fast-commit transaction commits or aborts, the related metadata is removed.

4.1.3 Clean Phase. After entering the clean phase, the flushing job scans the memtable to remove obsolete versions. For each key, it traverses the version chain up to the same tail version previously determined by `min_start`. If a tail version is found for a key, it has already been copied to the corresponding immutable memtable (either committed or uncommitted), and all versions older than the tail version are no longer visible to any transaction. Therefore, the flushing job can safely detach the tail version and all preceding versions from the chain and mark them as removed.

4.1.4 Concurrent Execution. The need to identify committed and uncommitted versions during the flush operation can increase the flush duration due to the additional processing involved. To maintain high flushing performance, we allow concurrent execution of multiple flushing jobs. To support this, we introduce a *collected* flag for each version in the memtable during the collection phase, which records the ID of the flushing job that collected it. This flag ensures that only one flushing job can collect a given version, even when multiple flushing jobs are running concurrently.

4.2 Version Management in the Disk Component

Since the version metadata required for transaction processing (such as timestamps for each version) is retained in memory, the on-disk SSTables store only raw key-value pairs along with a per-file timestamp. This per-file timestamp is only required for SSTables in L_0 , as files in this level may contain overlapping keys. Except for files in the stash region, all versions stored on disk are committed.

This design simplifies both transaction access to on-disk versions and compaction. Specifically, when accessing a version in L_0 , the per-file timestamp helps determine the most recent version when a key appears in multiple files. If two files have the same per-file timestamp (one generated from the committed immutable memtable and the other from the uncommitted immutable memtable), the version in the file generated by the uncommitted immutable memtable is considered more recent. This is because, for the uncommitted immutable memtable, the timestamp indicates that all versions were created by a fast-commit transaction that started at that timestamp. In contrast, for the committed immutable memtable, the timestamp represents an upper bound: all versions were created by transactions that committed before that time. For versions in other disk levels, timestamps are unnecessary because keys do not overlap within the same level, and version freshness is determined by the level order. This design enables SSTables to be compacted using conventional LSM-tree mechanisms, without storing per-version timestamps, except for those in L_0 created during memtable flushing.

4.3 Crash Recovery

When a fast-commit transaction commits, the metadata of its associated uncommitted files is moved from the stash region to L_0 , allowing these files to be managed by L_0 . After this, it generates a commit log and a checkpoint, which are then written to the WAL. With this design, crash recovery depends on the point of failure. If a crash occurs before the commit log is written, the system falls back to the previous checkpoint, and any uncommitted files are discarded, whether they are in the stash region or in L_0 . If the crash occurs after the commit log is written, the system uses the new checkpoint directly to perform recovery.

5 TRANSACTION PROTOCOL AND ISOLATION LEVEL

In our design and implementation, we adopt a variant of optimistic concurrency control (OCC) to implement MVCC [26] and explicitly disallow cascading aborts. Specifically, our approach guarantees snapshot isolation (SI) for all transactions. Under SI, unlike classic OCC for serializable transactions, there is no separate validation phase [1, 26]. Instead, conflicts are detected during write operations: the system follows a first-writer-wins policy for write-write conflicts and immediately aborts the conflicting transaction. Based on the same mechanism, Swan can also support read committed by creating the timestamp and snapshot at the beginning of each read operation rather than at transaction start. Serializable isolation is not supported in the current implementation. However, it could be implemented by adding additional metadata (e.g., read sets) and a validation phase to detect read-write conflicts and phantoms.

6 EVALUATION

In this section, we evaluate Swan to demonstrate the following advantages: (1) Leveraging its hybrid MVCC design, Swan avoids out-of-memory errors and achieves up to a 19% improvement in insertion performance when handling large, write-intensive transactions compared to the in-memory MVCC scheme, primarily due to reduced commit overhead. (2) Swan delivers transaction processing performance comparable to the in-memory MVCC scheme in concurrent environments across most workloads, while outperforming the out-of-memory MVCC scheme significantly, particularly for read-intensive workloads. The performance advantage of Swan becomes more pronounced when transaction sizes are smaller, and workload contention is higher. (3) Swan achieves an effective balance between in-memory and out-of-memory MVCC using a hybrid execution model that leverages the strengths of both approaches, making it well-suited for mixed workloads.

6.1 Experimental Setup

6.1.1 Implementation. Although the hybrid MVCC scheme can potentially be integrated into existing transactional key-value stores such as RocksDB, these systems are highly complex and involve many components that affect overall performance, making it difficult to isolate the impact of the MVCC scheme itself. To address this, we implement a prototype key-value store from scratch in Rust based on the proposed design. The memtable uses a lock-free hash table, and the disk component follows the SSTable format and binary search index of RocksDB, but without storing per-key version information (referred to as internal keys in RocksDB). Bloom filters are enabled, and the default block size for SSTables is set to 4 KB. In Swan, we reserve four threads for background flushing and one thread for background compaction. By default, compaction is disabled to eliminate its impact on performance.

6.1.2 Workload. To the best of our knowledge, there is no open-source transaction benchmark specifically designed for LSM-tree-based key-value stores. Therefore, following prior MVCC work [50], we modified the YCSB benchmark [12] to model various OLTP workloads. In YCSB, each workload consists of a load phase followed by a run phase. The load phase populates an empty database instance by inserting random key-value pairs. In our evaluation, the key size is fixed at 24 bytes, and the value size is either 100 bytes (default) or 1 KB. The run phase simulates workloads with varying read-write ratios. Following prior work [50], we define four workload mixtures to evaluate different transaction patterns: 1) read-only (100% reads), 2) read-write (50% reads, 50% writes), 3) update-intensive (20% reads, 80% updates), and 4) read-intensive (80% reads, 20% updates). Operations access key-value pairs following a *Zipfian* distribution that is controlled by a parameter (θ) that affects the amount of contention (i.e., skew), where $\theta=1.0$ is the highest skew setting. Similar to the setup in [50], we use a high skew factor ($\theta=0.8$) by default. To support transaction processing, we modify YCSB to wrap one or more operations within a transaction.

6.1.3 Hardware. All experiments are conducted on a machine equipped with an Intel Xeon W-1390P 8-core CPU (16 hyper-threads) and 64 GB of DRAM. For storage, we use an ext4 file system on a 500 GB Western Digital Black SN850 NVMe SSD.

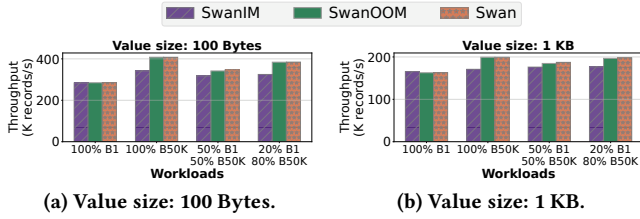


Figure 5: Insertion throughput.

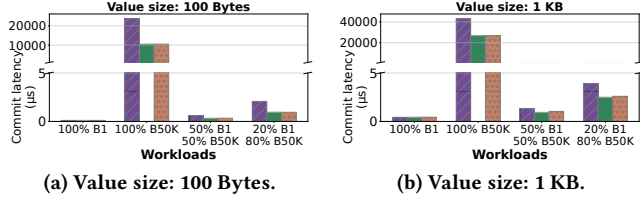


Figure 6: Commit latency.

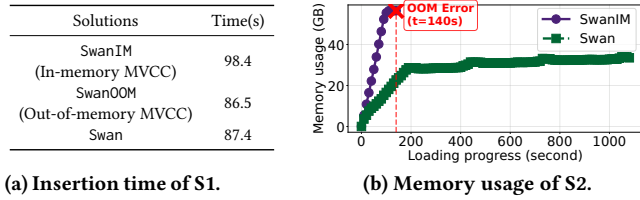


Figure 7: Large write transaction evaluation.

6.1.4 Baseline. We compare our hybrid MVCC solution, Swan, against two baseline solutions: SwanIM, which implements the in-memory (IM) MVCC scheme in our prototype, and SwanOOM, which implements the out-of-memory (OOM) MVCC scheme. All solutions use snapshot isolation as the isolation level.

6.1.5 Evaluation Methodology. The hybrid MVCC scheme in Swan combines the benefits of in-memory and out-of-memory approaches. Our end-to-end evaluation consists of two parts to showcase its advantages. First, in-memory MVCC handles concurrent transactions efficiently but struggles with large write transactions. Swan addresses this limitation by using the out-of-memory MVCC scheme for large write transactions. To evaluate this capability, we compare Swan with the in-memory MVCC scheme by varying transaction sizes in the load phase of YCSB. We record throughput, commit latency, loading time, and memory usage, as detailed in Subsection 6.2. Second, although the out-of-memory MVCC scheme effectively handles large write transactions, its performance is suboptimal in highly concurrent environments. Swan mitigates this issue by employing the in-memory MVCC scheme for concurrency-heavy workloads. We evaluate this strategy in Subsection 6.3 by running four different workloads during the run phase of YCSB and comparing the transaction processing throughput of three solutions. We also evaluate the impact of transaction size and *Zipfian* skew on performance. In addition, we conduct a microbenchmark in Subsection 6.4.

6.2 Write Transaction Evaluation

In this evaluation, we assess Swan to demonstrate its effectiveness and efficiency in handling large write transactions under different transaction sizes and memory conditions.

6.2.1 Transaction Size. We begin by evaluating insertion performance using different transaction sizes during the load phase to assess the impact of write transaction size on various MVCC schemes. A single thread executes write transactions with either a small batch size (i.e., 1) or a large batch size (i.e., 50,000) to insert 10 million key-value pairs. We vary the fraction of data inserted by small and large write transactions to create four workloads: 1) *100% B1* (all key-value pairs inserted with small write transactions), 2) *100% B50K* (all key-value pairs inserted with large write transactions), 3) *50% B1, 50% B50K* (half of the data inserted with small write transactions and half with large write transactions), and 4) *20% B1, 80% B50K* (20% of data inserted with small write transactions and 80% with large write transactions). Memtable flushing is disabled in this experiment to eliminate its impact on the results.

The insertion performance of the three solutions is shown in Figure 5, from which several observations can be made. First, when the transaction size is small, there is little to no performance difference among the three solutions. Second, with large transaction sizes, the out-of-memory MVCC scheme demonstrates clear benefits. For example, in the *100% B50K* workload, it achieves up to a 19% improvement in throughput, as shown in Figure 5a. This improvement is primarily due to reduced commit overhead (e.g., avoiding the need to update timestamps during commit). As illustrated in Figure 6a, the out-of-memory MVCC scheme can reduce commit latency by up to 55% in the *100% B50K* workload. Third, Swan and SwanOOM exhibit similar performance, as Swan leverages the out-of-memory MVCC scheme during this phase. Both outperform SwanIM when the transaction size is large, due to the reduced commit overhead described above. Fourth, when comparing value sizes of 100 bytes and 1 KB, the performance gap between the in-memory and out-of-memory MVCC schemes widens as the value size decreases. This is because the relative cost of commit operations becomes more significant for smaller values.

6.2.2 Large Write Transaction. In certain cases, such as bulk loading in memory-constrained environments, large write transactions can exceed available memory. The in-memory MVCC scheme is inefficient here because it caches all data until commit. To evaluate Swan’s performance under these conditions, we simulate two scenarios during the load phase: 1) One thread loads 8 million key-value pairs in a single large transaction (transaction size = 8 million), while another thread loads 2 million key-value pairs using small transactions (transaction size = 10). 2) One thread loads 80 million key-value pairs in a single transaction with a size of 20 million. Scenario 1 (S1) simulates a setting where one client performs bulk loading while another executes small write transactions concurrently. Scenario 2 (S2) represents an extreme case where a single large transaction exceeds memory capacity, which may occur in resource-constrained environments. In this experiment, we enable memtable flushing with a flushing threshold of 200,000 key-value pairs (approximately 200 MB). The value size is set to 1 KB.

As shown in Figure 7a, both Swan and SwanOOM achieve up to an 11% reduction in loading time in Scenario 1. When handling an extremely large transaction (Scenario 2), SwanIM runs out of memory, as it must keep all uncommitted data in memory. In contrast, Swan flushes uncommitted data to disk, allowing it to complete the task using roughly half the memory, as shown in Figure 7b.

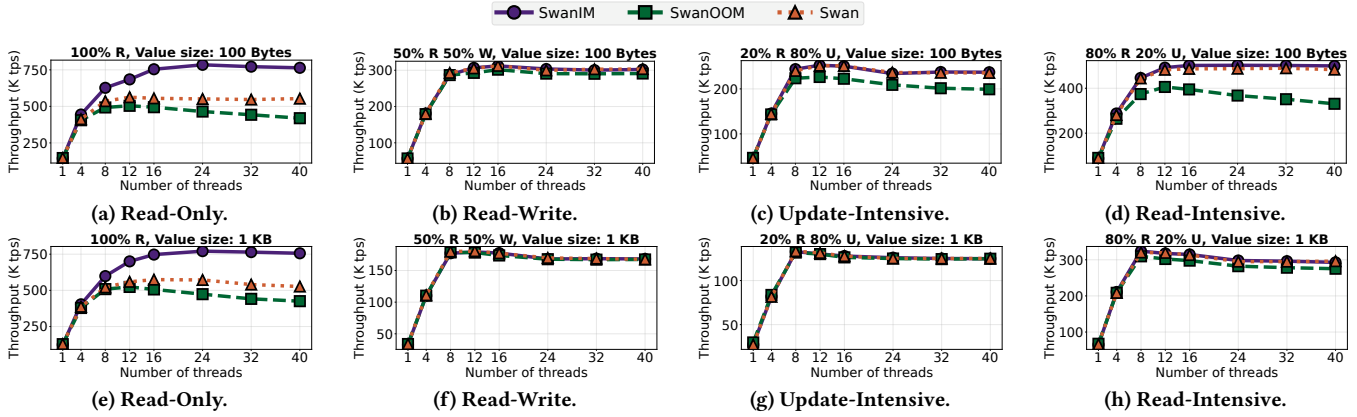


Figure 8: Transaction throughput under varying thread counts and workloads.

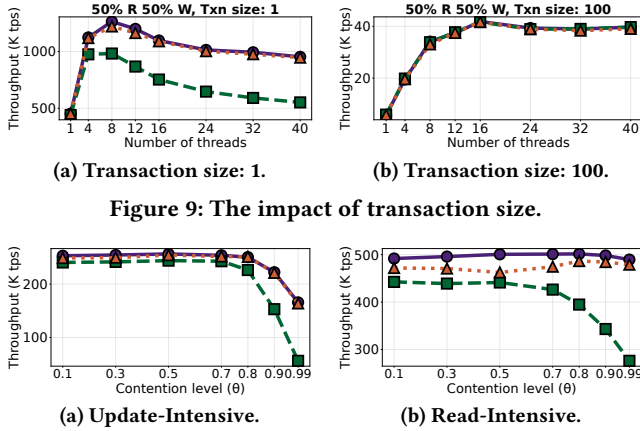


Figure 9: The impact of transaction size.

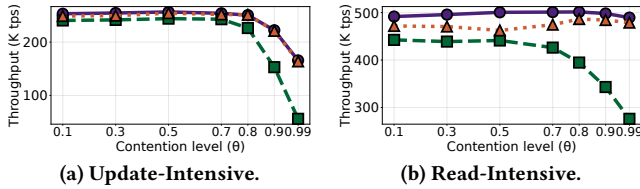


Figure 10: The impact of contention level.

6.3 Concurrent Environment Evaluation

A drawback of out-of-memory MVCC is its performance degradation under high concurrency due to expensive visibility checks and contention on shared data structures. Swan addresses this by using out-of-memory MVCC only for large write transactions and switching to in-memory MVCC for smaller, concurrent transactions to improve performance. To evaluate this hybrid approach, we run the load phase with the out-of-memory MVCC scheme, then execute four workload mixtures in the run phase using the in-memory MVCC scheme with varying numbers of concurrent threads. We compare Swan against two baselines: SwanIM, which uses the in-memory MVCC scheme throughout, and SwanOOM, which uses the out-of-memory scheme throughout. All experiments load 10 million key-value pairs and run 10 million operations per workload, with each transaction containing 10 operations. We disable memtable flushing to avoid its effect.

6.3.1 Transaction Processing Performance. Several observations can be made from the experimental results shown in Figure 8. First, in the read-write, update-intensive, and read-intensive workloads, Swan achieves performance comparable to SwanIM, as it executes concurrent transactions using the in-memory MVCC scheme. Both outperform SwanOOM, particularly for the read-intensive workloads.

Second, as the number of concurrent threads increases (i.e., under higher concurrency), the performance gap between Swan and SwanOOM widens. SwanOOM suffers more from thread contention. For example, as shown in Figure 8d, SwanOOM reaches peak throughput at 12 threads, while Swan continues to scale up to 16 threads. Third, for the read-only workload, Swan performs worse than SwanIM but still outperforms SwanOOM. This is because Swan executes transactions using in-memory MVCC and therefore, unlike SwanOOM, has no concurrent fast-commit transactions during the run phase, reducing contention on internal structures and the overhead of visibility checks. However, Swan must still apply the out-of-memory MVCC logic to verify the visibility of versions created under that scheme, introducing additional overhead compared to SwanIM. In contrast, for workloads with fewer reads (e.g., update-intensive), Swan achieves performance comparable to SwanIM. Finally, as the value size increases, the performance gap between SwanOOM and Swan narrows. This is expected, as larger value sizes require more time for reading and serialization/deserialization, which reduces the number of transactions executed in a given time frame and, in turn, lowers contention on internal data structures.

Overall, in concurrent environments, Swan shows performance improvements, achieving up to 32%, 3%, 15%, and 46% higher throughput than SwanOOM under read-only, read-write, update-intensive, and read-intensive workloads, respectively. Note that for the read-write workload, the performance improvement is limited because write operations require few or no visibility checks and, when multiple writes are performed within a transaction, incur longer execution time than read operations. This results in fewer transactions per unit time and reduced contention. However, as shown in Figure 9a, when the transaction size is smaller, Swan and SwanIM achieve greater performance gains on this workload, as smaller transactions increase the number of transactions per unit time and thereby amplify contention on shared internal data structures.

6.3.2 Transaction Size. Transaction size impacts performance by affecting the number of transactions and, consequently, the level of contention. To illustrate this, we compare transaction sizes of 1 and 100 under the read-write workload, as shown in Figure 9. Smaller transactions lead to a higher number of transactions executed per unit time, which in turn increases contention on internal

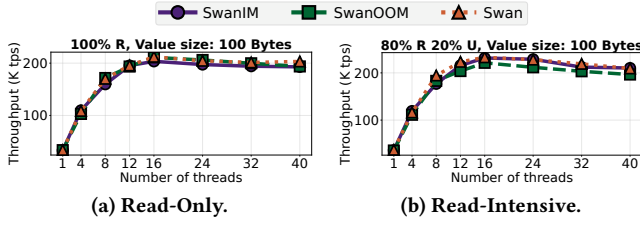


Figure 11: On-disk performance.

data structures. As a result, throughput with small transactions peaks at a lower thread count (e.g., 8 threads) and degrades beyond that. In contrast, larger transactions reach peak throughput at 16 threads, which then remains stable. Additionally, as transaction size increases, the performance gap between in-memory and out-of-memory MVCC narrows, since fewer transactions are processed per unit time, leading to reduced contention. For example, as shown in Figure 9b, the out-of-memory scheme achieves performance similar to the in-memory scheme on this workload.

6.3.3 Workload Contention. We compare the solutions under different levels of contention. We fix the number of threads to 16 and use the read-intensive and update-intensive workloads, with each transaction consisting of 10 operations. For each workload, we vary the contention level (θ) in the transactions’ access patterns. As workload contention increases, the likelihood of transaction aborts rises, which degrades performance. As shown in Figure 10, the performance of all three solutions decreases, with the out-of-memory MVCC scheme suffering more severely in high-contention environments. For example, as shown in Figure 10a, when the contention level increases from 0.7 to 0.9, the throughput of Swan drops by 12%, while SwanOOM experiences a 37% reduction.

6.3.4 On-Disk Performance. In the previous evaluation, we examined the hybrid MVCC scheme for in-memory versions. For committed versions on disk, freshness is determined by their level instead of timestamps, following conventional LSM-tree design, regardless of the MVCC scheme used.

To evaluate the performance differences among the three solutions when accessing on-disk versions, we enable memtable flushing by setting the threshold to 2,000,000 key-value pairs. After completing the load phase, we flush all in-memory data to the disk and perform compaction. All three solutions generate the same number of SSTable files after the load phase. Figure 11 presents the throughput of two read-oriented workloads (read-only and read-intensive) executed on this on-disk data. As shown, Swan, SwanIM, and SwanOOM achieve very similar performance on both workloads. This behavior is expected because, in these read-oriented workloads, the accessed versions reside on disk and are already committed. Consequently, version selection follows the standard LSM-tree read path, where freshness is determined by level ordering rather than MVCC visibility checks. As a result, MVCC-related overhead is bypassed for these workloads.

6.4 Microbenchmark

In this subsection, we evaluate the impact of fast-commit thread count, the fraction of in-memory MVCC versions, and the effectiveness of our concurrent flushing mechanism on performance.

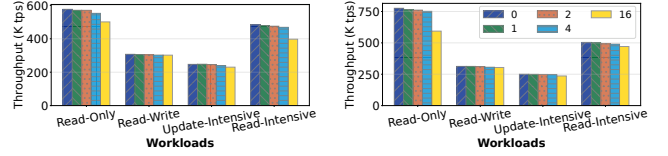


Figure 12: The impact of the number of fast-commit threads.

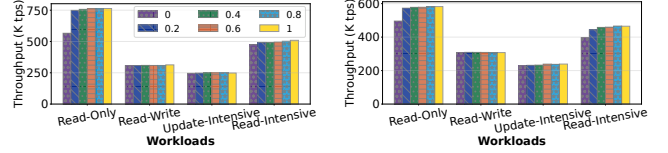


Figure 13: The impact of the fraction of IM MVCC versions.

6.4.1 The Impact of the Number of Fast-Commit Threads. Fast-commit transactions use out-of-memory MVCC for version management. Under high concurrency, contention on shared data structures for visibility checks can degrade performance. In this experiment, we fix the total number of threads at 16 and vary the number of fast-commit threads during the run phase to evaluate their impact on performance. During the load phase, data is inserted using either normal or fast-commit transactions. The remaining experimental setup follows Subsection 6.3.

As shown in Figure 12, increasing the number of fast-commit threads from 0 to 16 leads to performance degradation across all four workloads, primarily due to contention introduced by fast-commit transactions. This trend holds regardless of whether the versions are managed by the in-memory (IM) or out-of-memory (OOM) MVCC scheme. However, the performance degradation is relatively minor when only one or two fast-commit threads are used. For example, as shown in Figure 12b, in the read-only workload, using two fast-commit threads results in only a 3% performance drop compared to the baseline with no fast-commit threads. In contrast, using 16 fast-commit threads leads to a 22% drop. Based on this observation, a key advantage of Swan is its ability to selectively assign only one or two threads to handle large transactions using fast-commit, thereby minimizing their impact on overall performance.

6.4.2 The Impact of the Fraction of In-Memory MVCC Versions. In Swan, versions created by in-memory and out-of-memory MVCC can coexist. In this experiment, we evaluate how the fraction of in-memory MVCC versions affects performance. During the load phase, we vary the fraction of data created by normal transactions (which use in-memory MVCC) across six levels: 0, 0.2, 0.4, 0.6, 0.8, and 1. During the run phase, we use 16 concurrent threads to execute four workloads with either normal or fast-commit transactions. The rest of the experimental setup follows that described in Subsection 6.3.

From the experimental results in Figure 13, we make the following observations. The fraction of in-memory versions has a greater impact on workloads with more read operations. For example, as shown in Figure 13a, in the read-only workload, when the fraction of in-memory MVCC versions is 0 (i.e., all data are out-of-memory MVCC versions), the throughput shows noticeable

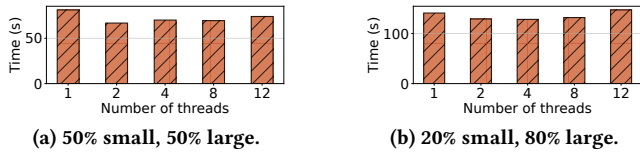


Figure 14: Evaluation of concurrent flushing.

degradation (up to 25%) compared to the case where the fraction is 1. This is because read operations incur higher visibility check overhead when accessing versions managed by the out-of-memory MVCC scheme. This degradation occurs regardless of whether the accessing transaction is a normal or fast-commit transaction, due to the visibility check overhead introduced by the out-of-memory MVCC scheme. On the other hand, for write-heavy workloads (e.g., read-write or update-intensive), the fraction of in-memory versions has little impact, as write operations do not access existing versions and incur no visibility check overhead.

6.4.3 Concurrent Flushing. To improve memtable flushing performance, we enable multiple flushing threads to run concurrently, allowing data in the active memtable to be flushed to disk more efficiently before the memtable grows too large. To evaluate it, we insert 10 million key-value pairs during the load phase while varying the number of flushing threads. Each value is 1 KB in size, and the flushing threshold is set to 200,000 entries (approximately 200 MB). We consider two flushing scenarios by varying the fraction of data inserted by fast-commit transactions (with a transaction size of 500,000), while the remaining data are inserted by normal transactions (with a transaction size of 1). In the first scenario, one thread inserts 50% of the data using normal transactions, while another thread inserts the remaining 50% using fast-commit transactions. In the second scenario, 20% of the data is inserted using normal transactions and the remaining 80% using fast-commit transactions. Figure 14 presents the data loading time for both scenarios under different numbers of flushing threads. We observe that concurrent flushing can reduce loading time by 8% to 18% compared to single-threaded flushing when using an appropriate number of threads (e.g., two threads in Figure 14a and four threads in Figure 14b).

7 RELATED WORK

MVCC Optimization. MVCC has been extensively studied in both main-memory and disk-based database systems [1, 9, 14, 15, 18, 19, 22, 26, 27, 38, 39, 50, 53–55]. Recent work has focused on optimizing MVCC for disk-based systems by improving commit protocols or leveraging transaction characteristics. Umbra [19, 35] optimizes MVCC for disk-based databases by keeping the versioning data of small transactions in memory, improving transaction processing performance. Furthermore, Umbra avoids re-stamping for large write transactions by introducing a single-writer mode. In contrast, our design addresses the same challenge without restricting the concurrency of write transactions. LeanStore [1, 29] optimizes the MVCC commit protocol to address scalability issues and reduce visibility check overhead in disk-based systems. It achieves this by introducing a fixed-size per-thread commit log. Similarly, Swan optimizes visibility checks, but it adopts a hybrid MVCC scheme to reduce their overhead while efficiently supporting large write transactions in LSM-tree-based systems.

Large Write Transactions. For certain transaction-supported key-value stores such as TiDB [20] and FoundationDB [56], the large write transactions are not well supported (e.g., the transaction limit is bounded by the memory allocated for the database in TiDB, and the transaction size cannot exceed 10,000,000 bytes of affected data in FoundationDB [2]). Although RocksDB adopts MVCC-based strategies (such as WriteUnprepared [41, 43]) to handle large write transactions, it struggles to deliver high performance under concurrent workloads. Swan uses a hybrid MVCC scheme to support large write transactions while preserving high performance under high concurrency. A recent work, Malt [16], also targets large write transactions in LSM-tree-based key-value stores. Like Swan, Malt allows uncommitted versions to be flushed to disk. However, the two systems tackle different challenges. Malt observes that large write transactions require excessive WAL space to track transaction states. To reduce WAL usage, Malt stores transaction states in an external structure. In contrast, Swan addresses the scalability problems of out-of-memory MVCC under high concurrency through a hybrid MVCC scheme.

Memory Management in LSM-Tree-Based Key-Value Stores. Efficient memory management is critical to the performance of LSM-tree-based key-value stores [4, 8, 10, 21, 31, 46]. For example, Triad [4] observes that many key-value store workloads exhibit skewed data popularity, where a small set of hot keys are updated far more frequently than cold keys. Based on this observation, Triad separates hot and cold keys during memtable flushing and flushes only cold keys to disk. However, Triad does not consider transactional semantics. In contrast, Swan separates versions based on their commit status. Our technique is also compatible with hot-cold separation strategies. For example, we can retain more recent data in the memtable by adjusting the value of `min_xip` during the collect phase. During the collect-and-clean phase, Swan scans the active memtable to identify versions based on their commit status. This process, particularly the clean phase, resembles garbage collection in multi-version systems [6, 14, 23, 28, 40, 48]. However, unlike traditional garbage collection, which removes versions no longer visible to active transactions, Swan focuses on distinguishing between committed and uncommitted versions. This allows Swan to keep uncommitted normal versions in memory while flushing both committed and uncommitted fast-commit versions to disk.

8 CONCLUSION

In this paper, we propose Swan, a hybrid MVCC scheme that combines in-memory and out-of-memory MVCC approaches, designed for LSM-tree-based key-value stores to efficiently handle large write transactions and support high-performance transaction processing in concurrent environments. We also introduce optimized memory management techniques to enable efficient and lightweight integration with existing LSM-tree components. Evaluation using the YCSB benchmark shows that Swan effectively combines the strengths of both MVCC schemes and delivers improved performance.

ACKNOWLEDGMENTS

The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 14202123, GRF 14200224).

REFERENCES

- [1] Adnan Alhomssi and Viktor Leis. 2023. Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1426–1438.
- [2] Apple Inc. 2026. Known Limitations. <https://apple.github.io/foundationdb/known-limitations.html>. Accessed: May 16, 2026.
- [3] David F Bacon, Nathan Bales, Nico Bruno, Brian F Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, et al. 2017. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*. 331–343.
- [4] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 363–375.
- [5] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency Control and Recovery in Database Systems*. Vol. 370. Addison-wesley Reading.
- [6] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *Proceedings of the VLDB Endowment* 13, 2 (2019), 128–141.
- [7] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.
- [8] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*. 275–290.
- [9] Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M Frans Kaashoek, and Nickolai Zeldovich. 2023. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 871–886.
- [10] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 17–32.
- [11] Audrey Cheng, Xiao Shi, Aaron Kabenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, et al. 2022. TAOBench: An End-to-End Benchmark for Social Network Workloads. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1965–1977.
- [12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. 143–154.
- [13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google’s Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.
- [14] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwillig. 2013. Hekaton: SQL Server’s Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1243–1254.
- [15] Jose M Faleiro and Daniel J Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1190–1201.
- [16] Chenguang Fang, Chen Qian, Qi Yang, Zeyu Wang, Zhenkun Yang, Fanyu Kong, Quanqing Xu, Hui Cao, Fusheng Han, and Chuanhui Yang. 2025. MaLT: A Framework for Managing Large Transactions in OceanBase. In *Companion of the 2025 International Conference on Management of Data*. 378–390.
- [17] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. *ACM SIGMOD Record* 40, 4 (2012), 45–51.
- [18] Nuno Faria, José Pereira, Ana Nunes Alonso, Ricardo Vilça, Yunus Koning, and Niels Nes. 2023. TiQuE: Improving the Transactional Performance of Analytical Systems for True Hybrid Workloads. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2274–2288.
- [19] Michael Freitag, Alfons Kemper, and Thomas Neumann. 2022. Memory-Optimized Multi-Version Concurrency Control for Disk-Based Database Systems. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2797–2810.
- [20] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: A Raft-based HTAP Database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.
- [21] Kecheng Huang, Zhaoyan Shen, Zhiping Jia, Zili Shao, and Feng Chen. 2022. Removing Double-Logging with Passive Data Persistence in LSM-tree based Relational Databases. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 101–116.
- [22] Jongbin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2020. Long-lived Transactions Made Less Harmful. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 495–510.
- [23] Jongbin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2021. Rethink the Scan in MVCC Databases. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 938–950.
- [24] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMLA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. 1675–1687.
- [25] Christoph Lameter. 2005. Effective Synchronization on Linux/NUMA Systems. In *Elato Conference*, Vol. 2005.
- [26] Per-Ake Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwillig. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proceedings of the VLDB Endowment* 5, 4 (2011), 298–309.
- [27] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krueger, and Martin Grund. 2013. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.* 36, 2 (2013), 28–33.
- [28] Juchang Lee, Hyungyu Shin, Chang Gyo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. 1307–1318.
- [29] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4536–4545.
- [30] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.
- [31] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. 2016. Lifetime-Based Memory Management for Distributed Data Processing Systems. *Proceedings of the VLDB Endowment* 9, 12 (2016), 936–947.
- [32] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook’s Social Graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.
- [33] C Mohan, Hamid Pirahesh, and Raymond Lorie. 1992. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. *ACM SIGMOD Record* 21, 2 (1992), 124–133.
- [34] MySQL. 2026. The InnoDB Storage Engine. <https://dev.mysql.com/doc/refman/8.4/en/innodb-storage-engine.html>. Accessed: May 16, 2026.
- [35] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research (CIDR)*.
- [36] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.
- [37] Andrew Pavlo and Matthew Aslett. 2016. What’s Really New with NewSQL? *ACM SIGMOD Record* 45, 2 (2016), 45–55.
- [38] Dan RK Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1850–1861.
- [39] PostgreSQL. 2026. PostgreSQL: The World’s Most Advanced Open Source Relational Database. <https://www.postgresql.org/>. Accessed: May 16, 2026.
- [40] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2023. One-shot Garbage Collection for In-memory OLTP through Temporality-aware Version Storage. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.
- [41] RocksDB. 2026. Transactions: TransactionDB and OptimisticTransactionDB. <https://github.com/facebook/rocksdb/wiki/Transactions>. Accessed: May 16, 2026.
- [42] RocksDB. 2026. WritePrepared Transactions. <https://github.com/facebook/rocksdb/wiki/WritePrepared-Transactions>. Accessed: May 16, 2026.
- [43] RocksDB. 2026. WriteUnprepared Transactions. <https://github.com/facebook/rocksdb/wiki/WriteUnprepared-Transactions>. Accessed: May 16, 2026.
- [44] Subhadeep Sarkar, Dimitris Staratzis, Ziehen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2216–2229.
- [45] Utku Sirin, Sandhya Dwarkadas, and Anastasia Ailamaki. 2021. Performance Characterization of HTAP Workloads. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1829–1834.
- [46] Hui Sun, Deyan Kong, Song Jiang, Yinliang Yue, and Xiao Qin. 2024. TrieKV: A High-Performance Key-Value Store Design With Memory as Its First-Class Citizen. *IEEE Transactions on Parallel and Distributed Systems* 35, 12 (2024), 2479–2496.
- [47] Hengrui Wang, Jiansheng Qiu, Fangzhou Yuan, and Huanchen Zhang. 2025. Rethinking The Compaction Policies in LSM-trees. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.
- [48] Yuanhao Wei, Guy E Blelloch, Panagiota Fatourou, and Eric Ruppert. 2023. Practically and Theoretically Efficient Garbage Collection for Multiversioning. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 66–78.
- [49] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Elsevier.

- [50] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.
- [51] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Gui Huang, Xinjun Yang, Wei Cao, and Feifei Li. 2021. Revisiting the Design of LSM-tree Based OLTP Storage Engine with Persistent Memory. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1872–1885.
- [52] Yang Guo, Jin Xue, and Zili Shao. 2026. The source code of Swan. <https://github.com/LAccordeur/swan/>. Accessed: May 16, 2026.
- [53] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proceedings of the VLDB Endowment* 8, 3 (2014), 209–220.
- [54] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. 1629–1642.
- [55] Ling Zhang, Matthew Butrovich, Tianyu Li, Andrew Pavlo, Yash Nannapaneni, John Rollinson, Huanchen Zhang, Ambarish Balakumar, Daniel Biales, Ziqi Dong, et al. 2021. Everything is a Transaction: Unifying Logical Concurrency Control and Physical Data Structure Maintenance in Database Management Systems. In *Conference on Innovative Data Systems Research (CIDR)*.
- [56] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 2653–2666.