Removing Double-Logging with Passive Data Persistence in LSM-tree based Relational Databases

Kecheng Huang, Shandong University, *The Chinese University of Hong Kong;* Zhaoyan Shen and Zhiping Jia, *Shandong University;* Zili Shao, *The Chinese University of Hong Kong;* Feng Chen, *Louisiana State University*

https://www.usenix.org/conference/fast22/presentation/huang

This paper is included in the Proceedings of the 20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

Removing Double-Logging with Passive Data Persistence in LSM-tree based Relational Databases

Kecheng Huang[†‡], Zhaoyan Shen[†*], Zhiping Jia[†], Zili Shao[‡] and Feng Chen[§]

[†]Shandong University, [‡]The Chinese University of Hong Kong and [§]Louisiana State University

Abstract

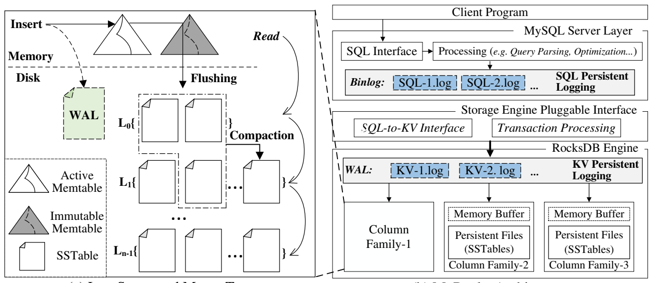
Storage engine is a crucial component in relational databases (RDBs). With the emergence of Internet services and applications, a recent technical trend is to deploy a Log-structured Merge Tree (LSM-tree) based storage engine. Although such an approach can achieve high performance and efficient storage space usage, it also brings a critical *double-logging* problem—In LSM-tree based RDBs, both the upper RDB layer and the lower storage engine layer implement redundant logging facilities, which perform synchronous and costly I/Os for data persistence. Unfortunately, such "double protection" does not provide extra benefits but only incurs heavy and unnecessary performance overhead.

In this paper, we propose a novel solution, called *Passive Data Persistence Scheme* (PASV), to address the double-logging problem in LSM-tree based RDBs. By completely removing Write-ahead Log (WAL) in the storage engine layer, we develop a set of mechanisms, including a passive memory buffer flushing policy, an epoch-based data persistence scheme, and an optimized partial data recovery process, to achieve reliable and low-cost data persistence during normal runs and also fast and efficient recovery upon system failures. We implement a fully functional, open-sourced prototype of PASV based on Facebook's MyRocks. Evaluation results show that our solution can effectively improve system performance by increasing throughput by up to 49.9% and reducing latency by up to 89.3%, and it also saves disk I/Os by up to 42.9% and reduces recovery time by up to 4.8%.

1 Introduction

Since 1970s, relational database (RDB) has been playing a central role in the heart of enterprise systems. The storage engine, as a core component in RDBs, typically adopts a B-tree based structure, which has been heavily tuned and optimized for traditional database workloads, such as online transaction processing and online analytical processing.

[*]Corresponding author



(a) Log-Structured Merge Tree (b) MyRocks Architecture

Figure 1: Architecture of a typical LSM-tree based RDB.

With the emergence of Internet services and applications, the classic B-tree based storage engine, after dominating database systems for decades, is facing several critical challenges. Unlike conventional database workloads, these new applications and their supporting systems often generate very write-intensive workloads [1]. Many of them use relatively simple and fixed data schema [2]. Some systems adopt expensive flash storage [3–9], and thus they are very sensitive to storage space usage and demand efficient data compression for cost saving. Correspondingly, the storage engine design must meet a set of new requirements—scalability, space efficiency, compressibility, I/O sequentiality, etc.

To address these new challenges and demands, a recent technical trend is to deploy a *Log-structured Merge Tree* (LSM-tree) based storage engine [10–18] in RDBs. A typical example is Facebook's MyRocks [7]. Different from the traditional structure of MySQL, MyRocks replaces the original B-tree based storage engine, InnoDB [19], with an LSM-tree based storage engine, RocksDB [12]. Although such an LSM-tree based storage engine significantly outperforms the B-tree based engine in terms of both performance and storage space usage, it brings a critical new issue, which can incur heavy and unnecessary performance overhead.

As illustrated in Figure 1, integrating an LSM-tree based storage engine in an RDB essentially creates a *two-layer structure*: (1) On the top RDB layer, the RDB logic handles the database related complexities, such as buffer pool management, query optimization, SQL query processing, transaction support, data recovery, etc; (2) At the bottom storage engine layer, the storage engine processes requests from the RDB layer and is responsible for reliably and efficiently storing data in persistent storage. Such a design enables great flexibility, efficiency, and portability, allowing the two layers to be independently optimized without affecting each other. However, as a complete data store itself, the LSM-tree based storage engine, such as RocksDB [12], has many functions similar to the RDB atop it. Some of these functions are *redundant* and *unnecessary*, which can cause severe resource waste and negative performance impact. One such critical component is *log*, which is the focus of this paper.

Double-logging problem. In an RDB system, a *binlog* records all processed SQL statements. Once system crash happens, the SQL statements in the binlog are replayed for data recovery. With an LSM-tree based storage engine, each SQL statement is translated into a sequence of key-value items (KVs), which are stored in the underlying LSM-trees for persistent storage. In an LSM-tree based storage engine, a *Write-ahead Log* (WAL) is maintained to record all KV update operations. Each KV must be first written to the WAL before being inserted into the tree structure, which is also for the purpose of data recovery upon system crash. In the whole stack, any change made to the database is "protected" twice—one redundant copy is preserved in the database's binlog and another one is in storage engine's WAL. Such redundancy apparently leads to unnecessary space usage, but even worse, these log-related I/O operations are synchronous and reside in the system's critical path, resulting in significant, needless I/O overhead and severely affecting system performance.

We call the above-said issue a *double-logging problem*, which is a system situation that data is over-protected by preserving database changes multiple times more than necessary. To the best of our knowledge, this is the first time this problem is revealed in RDBs with LSM-tree based storage engines.

It is worth noting that the double-logging problem is *not* the "log-on-log" problem [20], which typically appears when running a log-structured file system on a log-structured flash FTL. In the log-on-log problem, the upper-level log is stored on another lower-level log structure. In the double-logging problem, in contrast, the two logs are independent and stored separately (the binlog is not stored in or relies on the WAL). Thus, the double-logging problem does not involve issues known in the log-on-log problem, such as data remapping, unaligned segments, and uncoordinated garbage collection, etc. Rather, it concerns more about the unnecessary redundancy in I/O operations and storage space usage.

Our solution. In order to address the double-logging problem, our key idea is to completely remove WAL from the LSM-tree based storage engine, and solely rely on binlog for data recovery. A naïve solution is to directly disable WAL (e.g., RocksDB provides a configurable option). However, the system integrity cannot be guaranteed due to the uncoordinated SQL and KV operations, which can cause incomplete or erroneous recovery. Even if we perform data recovery by replaying the binlog, we have to replay all the records in the entire binlog, one after another sequentially. This would incur an excessively time-consuming data recovery process, causing a long service outage and system downtime.

In this paper, we propose a novel solution, called *Passive Data Persistence Scheme* (PASV), to address these challenges. It includes three major components: (1) To bridge the semantic gap between the RDB layer and the storage engine layer, we create a special data structure, called *Flush Flag*, to deliver the critical RDB-layer semantics, including the critical data persistence point, each KV item's logical sequence number, etc. (2) To avoid intrusive changes to the current modular system design, we propose a *Passive Memory Buffer Flushing Policy* to pass a flush flag for each LSM-tree along with its regular flush operations in the storage engine. Without requiring to explicitly flush the memory buffers, we can avoid undesirable performance impact caused by flushes and minimize structural changes to the two layers. (3) We also develop an *Epoch-based Persistence* (EBP) policy to determine the global data persistence point, guaranteeing that we only need to perform *Partial Data Recovery* to recover the necessary data and eliminate redundant KV operations that have already been persisted before system crash.

We have implemented a fully functional, open-sourced prototype based on Facebook's MyRocks [8]. Our prototype involves minor changes (only about 500 lines of C/C++ code) and is publicly available [21]. Evaluation results based on LinkBench [1] and TPC-C [22] show that our solution can effectively increase throughput by up to 49.9% and reduce latency by up to 89.3%, and it also saves disk I/Os and recovery time by up to 42.9% and 4.8%, respectively.

The rest of the paper is organized as follows. Section 2 introduces the background. Sections 3 and 4 explain the problem and the challenges. Section 5 describes the design details. Section 6 gives the evaluation results. Section 7 discusses the related work. The final section concludes this paper.

2 Background

2.1 Log-structured Merge Tree

LSM-tree structure. LSM-tree adopts a unique append-only structure [10], which is specially tailored for handling intensive small KVs. In LSM-tree, the incoming small and random KVs are firstly buffered and sorted within a memory buffer, called *MemTable*. Once the MemTable is full, it is transformed into an immutable buffer and flushed to storage as an *SSTable*. SSTable is the basic storage unit in LSM-tree. Each SSTable stores its KVs in the order of the keys.

SSTables on storage are organized in a multi-level structure, each level of which, except the first level, maintains a sequence of SSTables with non-overlapping key ranges. Two different levels may have overlapping key ranges. A lower level typically maintains several times more SSTables (wider) than its adjacent upper level, forming a structure like a tree. If the number of SSTables at a level exceeds size limit, selected SSTables are merged into the lower level through merge sort, which is called *Compaction*. Upon a query, a binary search is performed, level by level from top to the bottom, until finding the item or returning "Not Found". Figure 1(a) illustrates the structure of a typical LSM-tree.

In order to prevent the loss of in-memory data upon system failures, all updates that are made to the memory buffer (a.k.a. MemTable) must be first written into an on-disk log structure, called *Write-ahead Log* (WAL) [23, 24]. When the system restarts after a crash, the records in the WAL are replayed to reconstruct data that are originally in the memory buffer.

Multi-LSM-tree based structure. Modern KV storage engines often maintain more than one LSM-tree to create I/O parallelism for high-speed storage devices, such as SSDs, to achieve better performance. Let us use RocksDB [12] as an example. In RocksDB, it maintains several so-called *Column Families* (CFs). Each column family corresponds to one LSM-tree [1]. Only one WAL is maintained for all LSM-trees, which is called *Group Logging* [25] or *Group Commit* [26]. Although keeping one WAL for all LSM-trees can bring performance advantages in transaction processing, it leads to an issue, which is that the batched KVs logged in WAL may be inserted into LSM-trees at different speeds, causing an inefficient recovery process upon failures. We will explain the problem in more details later in this paper.

2.2 LSM-tree based Storage Engine

A recent technical trend is to deploy LSM-tree based storage engine in RDBs. A typical example is Facebook's MyRocks [8], which adopts RocksDB as the storage engine in MySQL database. Next, we will first explain the benefits of using LSM-tree as storage engine for RDBs, and then discuss its structure and the inherent problem.

Benefits of LSM-tree as storage engine. There are two major advantages. First, the LSM-tree structure is known for its high performance under write-intensive workloads. In new system environments, such as Internet services, which need to handle huge traffic of constantly incoming data, the performance benefit of LSM-tree is particularly appealing. The second advantage is space efficiency. Traditional storage engines typically use a B-tree based structure. Having been heavily optimized for query performance, such storage engines typically demand more storage space for complex indexes and metadata maintenance. Along with inefficient compression, the disk space usage becomes a concerning issue [4, 6, 8]. In contrast, LSM-tree is a log-structured design, which persists data in an append-only way and stores data in a sorted manner. This allows data to be organized in a more condensed format in storage. Assuming each level is 10 times larger than the upper level, theoretically, the space amplification of LSM-tree structure can be limited at a low level (approximately 1.111... times of the original data size).

[1] In this paper, we use the two terms, column family and LSM-tree, interchangeably. Unless otherwise specified, they both refer to an LSM-tree structure in the multi-LSM-tree storage engine.

Table 1: Comparison between MyRocks and MySQL.

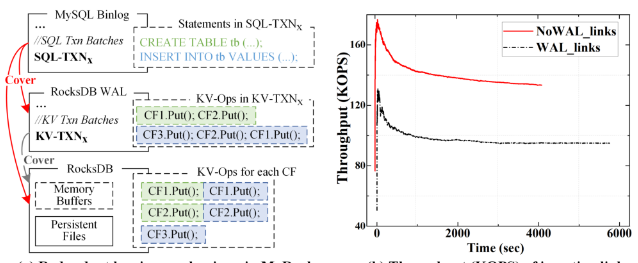
	Total Execution Time (Seconds)	Throughput (KOPS)	Occupied Disk Space (GB)
MyRocks	10,895.8	40.9	70.7
MySQL	13,584.7	32.8	108.6

In Table 1, we show the results of a preliminary test to illustrate the performance and space efficiency of MyRocks compared to MySQL (version 5.6) with InnoDB as the storage engine. Both MyRocks and MySQL use the default configurations. We measure their performance using the same 100GB UDB-style workload [2] generated by LinkBench [1] with one loader. More details about the system setup can be found in Section 6. We can see that MyRocks saves 34.9% disk space compared to MySQL and also substantially outperforms MySQL in both total execution time (19.8%) and throughput (24.7%). This result well demonstrates the performance and storage advantages of adopting an LSM-tree based storage engine in RDBs.

RDBs on LSM-tree. More recently, many RDBs begin to adopt LSM-tree based storage engines. For example, Spanner [5] is an LSM-tree based database, which is a full-featured SQL system for distributing data at global scale and supports distributed transactions [27]. Facebook's MyRocks [7] is a MySQL-based implementation for serving the UDB scenarios [6]. MyRocks replaces the original B-tree based storage engine with RocksDB. Some other databases also construct their storage engines based on the LSM-tree structure [28–30]. Below we use MyRocks as a representative example to explain the basic structure of an LSM-tree based RDB.

Figure 1(b) shows the architecture of MyRocks, which provides an SQL interface but adopts an LSM-tree based storage engine. It is a two-layer structure and has three major components: (1) a generic MySQL server layer, (2) a pluggable SQL-to-KV translator, and (3) an LSM-tree based storage engine layer. The MySQL server layer organizes user requests in SQL transactions, logs SQL statements in the *binlog*, and issues the transactions to the SQL-to-KV translator. The translator converts the SQL statements of each transaction into a *KV batch* [31], which consists of a set of KV items. The KV batch is then sent to the LSM-tree based storage engine, which issues KV items to the corresponding column families. In RocksDB, before inserting a KV item into the LSM-tree's memory buffer (MemTable), it is first written to the WAL. Once all the KV items belonging to a transaction are inserted into the MemTables, the transaction can be safely regarded as "persistent", and a commit flag for this transaction is returned.

For data recovery, a two-phase recovery process is performed. The storage engine first retrieves the batched KV items in the WAL and rewrites them into the MemTables of the corresponding LSM-trees. Then, the MySQL server layer replays all the transactions in the binlog after the safe point



(a) Redundant logging mechanisms in MyRocks **(b) Throughput (KOPS) of inserting links**

Figure 2: The double-logging problem in MyRocks.

(commit flag). The first phase guarantees that the storage engine itself is recovered to the state before the crash; The second phase guarantees the consistency of the SQL logic in the MySQL server layer.

3 The Double-Logging Problem

An LSM-tree based RDB essentially forms a two-layer structure. As shown in Figure 1(b), the RDB layer and the storage engine layer each maintains a set of complete logging mechanisms, individually, for data persistence and recovery. The two sets of mechanisms are independent of each other and co-exist in the system. Figure 2(a) illustrates the redundant functions for data persistence in this structure. We can see that despite the differences in the data persisted in binlog and WAL, the two logs are for the same purpose.

Interestingly, according to the end-to-end theory [32], such a "double protection" would not bring extra guarantees for data safety but only heavy and unnecessary performance penalties. First, all logging I/Os have to be performed twice. In the upper RDB layer, a transaction needs to be first written into the binlog in the form of SQL statements; In the lower storage engine layer, the KV items translated from the transaction need to be first written into the WAL in the form of KV operations. This is clearly a significant waste of storage I/O resources. Second, and more importantly, due to the requirement for safely committing a log record, the involved I/Os must be synchronous and performed in a serial manner to ensure correctness [33–35]. As a result, these redundant I/O operations are unfortunately in the critical path, which further amplifies the negative effect on system performance. We call it a *double-logging* problem. To the best of our knowledge, this paper is the first work unveiling this hidden, critical problem in LSM-tree based RDBs.

To illustrate the overhead, we perform a preliminary test on MyRocks. We turn off WAL in the RocksDB storage engine and keep the other configurations as default. We use LinkBench to generate a workload with around 437 million SQL requests based on Facebook's UDB distribution. Figure 2(b) shows the throughput of inserting links with 10 loaders. By simply disabling WAL, we can enhance the overall throughput (KOPS) by 44.6%. This result shows a great potential for performance improvement by solving the double-logging problem in LSM-tree based RDBs.

Another possible choice is to remove binlog in the upper RDB layer and rely on WAL for recovery. However, this approach is unreliable for two reasons. First, unlike binlog, the storage engine's WAL persists KVs individually, lacking sufficient semantic information for safe recovery. For example, a transaction written in the binlog is regarded as containing complete information at the SQL level. However, if a crash occurs in the middle, the KVs translated from this transaction could be partially persisted in the WAL. If we removed the binlog, the atomicity of such on-the-fly transactions could be compromised. Second, besides data recovery, the RDB's binlog also serves for other functions, such as instance replication and synchronization, which cannot be handled by solely keeping WAL. As indicated by the end-to-end philosophy [32], it is a more sensible choice to keep binlog rather than WAL to address the double-logging problem.

4 Critical Challenges

Our main idea is to remove WAL while still retaining data reliability upon failures. However, it is non-trivial to achieve this goal. We must address three critical challenges.

• *Unwarranted data persistence*. In an LSM-tree based RDB, the upper RDB layer translates each transaction into multiple KV items and submits to the lower storage engine layer, which receives the KV items and makes them persistent. It is assumed that the KV items are persisted once the RDB receives completion. However, if the WAL was eliminated, such an assumption could not hold anymore. In other words, the transaction commit flag in the binlog can no longer be reliably viewed as the safe point for data persistence, since the upper RDB layer cannot be certain whether the transactions prior to this point have been truly made persistent or not.

• *Partial persistence*. In a storage engine with multiple LSM-trees, an SQL transaction is translated into a batch of KV items, which are often distributed to multiple LSM-trees, a.k.a. *Column Families* (CFs) in RocksDB. Once the memory buffer (MemTable) of a CF is filled up, it is flushed to the storage in the form of an SSTable. Since the sizes and arrival rates of the KV items accommodated in different LSM-trees may vary, such memory buffer flushes can happen at distinct frequencies and are completely uncoordinated across the LSM-trees. This could lead to a situation that at a point of time when system failure happens, an SQL transaction may be partially persistent. In other words, some KV items of the transaction have already been flushed to storage but some others are not yet (still in the volatile memory buffers).

• *Lost track of LSN*. LSM-tree based RDBs use a *Log Sequence Number* (LSN) [36] for concurrency control and meeting the ACID requirements [24]. Each KV is allocated with an LSN, which is essentially a globally unique sequence number. The items of a KV batch, which corresponds to an SQL transaction, are guaranteed to receive a sequence of consecutive LSNs. With WAL, we can guarantee that each recovered KV item still carries the originally assigned LSN.

However, if the WAL was eliminated, we would lose track of these LSNs. Even replaying the binlog cannot regenerate the lost LSNs, and if we did so, the LSNs of KV items persisted on storage would become incomplete and out of order.

In the next section, we will present our design to handle these challenging issues. We develop a set of effective schemes for safely removing WAL while still fully retaining the guarantees for data persistence.

5 Design

In this paper, we present a highly efficient solution to address the double-logging problem in LSM-tree based RDBs. We desire to achieve three important goals in our design.

• *Aim #1: Effectiveness and efficiency*. Our solution should effectively and efficiently address the double-logging problem. We need to achieve not only low performance overhead during normal runs but also fast data recovery when failure happens.

• *Aim #2: Data persistence and correctness*. Removing the redundant logging should not come at the cost of weakening the promise for data persistence and correctness. Data reliability should remain identical to the existing system.

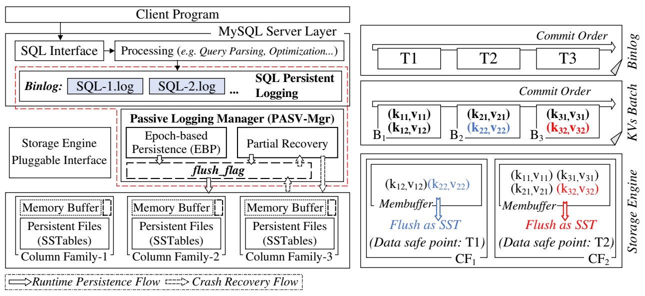
• *Aim #3: Minimal and non-intrusive changes*. A merit of the current LSM-tree based RDB design is its modularity. The RDB layer and the storage engine layer are relatively independent. We should avoid introducing complicated, intrusive changes and retain the current system's modular structure.

By following the above-said three design goals, we propose a *Passive Data Persistence Scheme* (PASV) to address the double-logging problem in LSM-tree based RDBs. We have implemented a full-featured, open-source prototype [21] based on Facebook's MyRocks. It is worth noting that the design rationale presented in this paper can also be applied to other LSM-tree based RDBs with similar double-logging problems. Although the detailed implementation may vary, our prototype provides guidance for eliminating the redundant logging for optimized performance while still achieving fast and reliable data recovery upon system failures. In the following, we will first introduce the overall design and then describe each component one by one.

5.1 Overview

Figure 3(a) illustrates the architecture of PASV for LSM-tree based RDBs. To minimize changes to the existing system design, we keep the original two-layer structure to the maximum, and only remove the Write-Ahead Log (WAL) in the lower storage engine layer and solely rely on the binlog in the RDB layer for data recovery.

We introduce three new components for the goal of eliminating redundant logging but still achieving reliable and efficient data recovery upon failures. In particular, (1) *Passive Logging Manager* (PASV-Mgr) coordinates the data logging and recovery operations between the two layers, and a special



(a) Architecture of PASV (b) Memory buffer flush operations

Figure 3: PASV architecture and data-flow.

Flush Flag is used to deliver the data recovery-related transaction information between the two layers, (2) *Epoch-based Persistence* (EBP) module determines the nearest global safe point for reliably and completely recovering the lost data upon failures, and (3) *Partial Recovery* process identifies the minimal number of KV items to be recovered for each column family, which enables fast and efficient data recovery. We introduce each component in details below.

5.2 Passive Data Persistence

In order to address the double-logging problem, our key idea is to completely remove the WAL in the lower storage engine layer and rely on the binlog in the upper RDB layer for data recovery. This is for two reasons.

First, the binlog contains a complete set of the original SQL transactions committed to the database, which makes it possible for us to recover all the database data in their original format. Second, all the KV items received at the lower layer can be reconstructed from the original transactions, even if the underlying storage engine cannot guarantee the data persistence. However, safely recovering all transaction data in their original order is non-trivial.

Challenges. The main difficulty stems from the uncoordinated flushes of the memory buffers of the underlying LSM-trees. As mentioned previously, modern LSM-tree based storage engine, such as RocksDB, maintains multiple LSM-tree structures for parallelizing I/Os and maximizing the achievable performance. Each LSM-tree, a.k.a. *Column Family* (CF), maintains an individual and independent memory buffer (MemTable) to temporarily hold incoming KV items. When the memory buffer reaches the size limit, it is flushed to disk or SSD for persistent storage. Without the WAL, upon a system failure, the KV items in the volatile memory buffer would be lost and unrecoverable. The problem is that the flush operations of memory buffers of different column families are completely independent and uncoordinated, meaning that memory buffer flushes may happen at different frequencies depending on the sizes and arrival rates of the incoming KV items, which often vary significantly across column families and dynamically change over time. As a result, the KV items translated from one SQL transaction may be persisted on storage at different time points. When a failure happens, we may have a *partially* persisted transaction and the transactions may not be fully persisted in their original serial order as their commit flags in the binlog.

Figure 3(b) illustrates such an example, in which we have two column families, CF_1 and CF_2, and three transactions, T_1, T_2, and T_3. If CF_2 flushes before CF_1, transaction T_3 would be fully persisted on storage, while T_1 and T_2 still have partial data, (K_{12}, V_{12}) and (K_{22}, V_{22}), in the volatile memory buffer of CF_1. If a failure happens, the two transactions (T_1 and T_2) would become incomplete on storage, and we can find that the transactions are not fully persisted in their original commit order in the binlog. In order to ensure complete data recovery, we would have to replay the entire binlog from the beginning, since we cannot determine which transactions are safely and completely persisted on storage.

An active approach. A simple solution to the above-said issue is to directly insert an *Active Flush Point* after each or a number of transactions to explicitly invoke the underlying storage engine layer to flush the memory buffers of all the LSM-trees at the same time, arbitrarily creating a synchronization point. Although this "active" approach guarantees that all transactions before the active flush point are persisted safely, it has several limitations. (1) The transactions are essentially serialized, which foils the effort of creating parallelism. (2) Frequent flushes would in effect invalidate the memory buffers, causing many small and synchronous I/Os to storage. (3) Most importantly, this approach impairs the effort in the current design for modularity. It forces the RDB layer to directly control the memory buffer operations at the lower storage engine layer, which we desire to avoid.

A passive approach. To avoid intrusive changes to the existing two-layer structure, we develop a "passive" approach to handle uncoordinated flushes in a more elegant way. Here is how it works. When the storage engine flushes a memory buffer, a special KV item, called *Flush Flag*, is inserted into the memory buffer and flushed together with other KVs to the storage. The purpose is to place a "marker" in the persistent storage to indicate the progress of the latest flush operation.

A flush flag is a special-purpose KV item. Its key is a randomly chosen 128-bit magic number, which indicates that this KV item contains a flush flag rather than user data. Each column family has a unique key for its flush flag. The value contains a vector of four metrics $< CF, TSN, LSN_{first}, LSN_{last} >$. *CF* is the column family (LSM-tree) whose memory buffer is being flushed; *TSN* is the *Transaction Sequence Number* of the last transaction whose KV items are inserted in the memory buffer of the column family; LSN_{first} and LSN_{last} are the LSNs of the first KV and the last persisted KV of the transaction, respectively. To retrieve the latest flush flag, we simply query the LSM-tree using the corresponding key, which is just like retrieving any regular KV item. In this way, we can use a flush flag to keep track of the latest transaction and its KV items that are persisted in storage during a flush operation, from which we can derive the safe point for data persistence during recovery.

This approach is safe due to the *serial property* of transaction processing in LSM-tree based RDBs. In LSM-tree based RDBs, it is guaranteed that the KV items are processed in a serial manner: (1) During the transaction commit process, all transaction records are persisted to the binlog in serial; (2) The SQL transactions are parsed in the RDB layer and translated into KV batches in the storage engine layer in serial; (3) The KV items that are translated from a transaction are inserted into the LSM-trees' memory buffers in serial. Hence we can ensure that all KV items logically prior to the last KV item of the last transaction in a column family would never be persisted to storage later than it. It is worth noting that this serial property is not unique to MyRocks. Other databases also adopt the serial design. For example, Amazon Aurora [34], a novel OLTP-oriented RDB, is known to "model the database as a redo log stream" and "exploit the fact that the log advances as an ordered sequence of changes".

Based on this serial property, we can conclude that for a column family CF_i in the LSM-tree storage engine, if the retrieved flush flag contains transaction TXN_p, the KV items of transaction TXN_{p-1} and transactions prior to it in CF_i must have already been persisted. Thus, transaction TXN_{p-1} can be regarded as the *Data Safe Point* of column family CF_i. Comparatively, transaction TXN_p may be partially persisted (some KV items of the same batch may not arrive in the memory buffer yet). Hence we call transaction TXN_p the *Data Persistence Point* of column family CF_i, indicating the current position of persisting data.

In the example of Figure 3(b), if the memory buffers of CF_1 and CF_2 are flushed, their data safe points are transaction T_1 and T_2, while their data persistence points are T_2 and T_3, respectively. We also see in this example that due to the different sizes and arrival rates of the involved KV items, the column families may make unequal "progresses" in terms of persisting data for transactions (T_2 vs. T_3 in this example). We will discuss how to address this problem in the next section.

This passive approach brings several important advantages. First, we can minimize intrusive changes to the existing modular design. The RDB layer does not need to explicitly invoke memory buffer flush operations in the storage engine layer. Instead, the flush flag is naturally persisted in storage together with other KV items when the memory buffer is flushed. Second, the memory buffer flushes still follow the original logic. We do not need to prematurely flush a memory buffer that is not full yet, which maximally retains the benefits of parallelism and memory buffering. Third, the memory buffer management also remains nearly unchanged, not incurring extra performance overhead. The flush flag is very small, meaning that the spatial overhead is also minimal.

5.3 Epoch-based Persistence

The storage engine has multiple column families (CFs), each of which is an individual LSM-tree with a volatile memory buffer. As mentioned above, different CFs may make unequal progresses in terms of persisting KV items of transactions. Thus, we need to determine the latest transaction whose KV items have been *fully* persisted on storage across all CFs.

Inspired by Epoch-based Reclamation [37–40], we propose an *Epoch-based Persistence* (EBP) to identify the global safe point for data persistence. The basic idea is to use *Local Epoch* to separately manage each CF's data safe point, and use *Global Epoch* to identify the global data safe point, which determines where we should start in binlog for recovery.

Local epoch. Flush flag maintains the last transaction and the last KV item being flushed to persistent storage, which is the *Local Data Persistence Point* as described in the previous section. The passive persistence manager, PASV-Mgr, tracks the progress of persisting data made by each column family by maintaining a tuple $< CF, TXN >$ for each column family. For a column family CF_i, we record the corresponding local data persistence point TXN_p, which is the transaction recorded in its flush flag. It indicates that all KV items of transactions prior to TXN_p in column family CF_i must have already been persisted safely on storage. Note that a "locally safe" transaction may not be safe in a "global" viewpoint, since some KV items of the transaction may not be persisted yet in another column family. A local epoch is the transactions between two consecutive persistence points in the binlog.

Global epoch. Based on the local epoch, the data persistence status for each column family (CF) is known by the system. That is, the system is aware of the local data persistence points for all CFs. Each time when a new local epoch is created, we can derive the *Global Data Persistence Point* by comparing the local data persistence points. The smallest TXN, or the earliest local data persistence point, is the global persistence point in the sequence of transactions. For a given global persistence point, all the transactions committed in the binlog prior to it must have already been safely and completely persisted on storage. If system crash happens, only the transactions starting from this global data persistence point (including itself) need to be examined and replayed.

Figure 4 illustrates an example, in which we can see that the local epochs indicate that column family CF_1 has made the most significant progress by flushing the KV items of all transactions prior to TXN_t, while CF_2 is the slowest one, which only flushes until transaction TXN_n. In terms of the global epoch, it is clear that the current global data persistence point is at transaction TXN_n, meaning that all transactions before TXN_n must have been completely persisted. Upon data recovery, we only need to replay transactions starting from transaction TXN_n and thereafter.

5.4 Partial Recovery

The epoch-based persistence policy enables us to quickly determine the latest transaction that is made completely persistent on storage. Upon a system failure, we need to recover data by replaying the transactions after that.

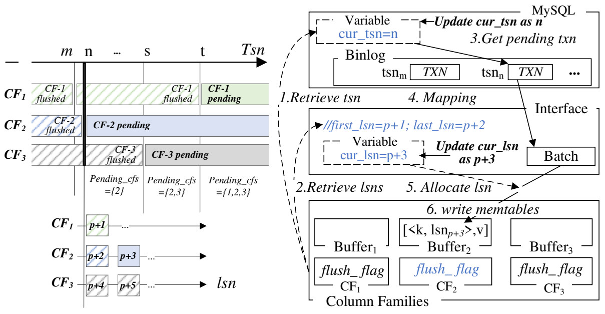


Figure 4: Partial recovery and LSN allocation.

A simple method is to replay all the KV operations in all column families. However, as illustrated in Figure 4, since column families make different progresses, such a conservative approach would be very inefficient and wasteful. For example, there is no need to reinsert the KV items of transaction TXN_n in column family CF_1, because the KV items of this transaction have already been flushed to storage. Thus, we develop *Partial Recovery* for column families whose local data persistence point differs from the global data persistence point. In other words, we selectively skip the transactions and KV items that have already been persisted.

Partial recovery consists of two steps. (1) We first retrieve the latest flush flag from each CF to determine the global data persistence point and the local data persistence points, using which we can determine the range of transactions that we need to replay for each CF. (2) Then we perform the replaying operations by scanning through the binlog and translating each SQL transaction to KV items and submit to the corresponding CFs. Due to the partial recovery policy, the KV items are only dispatched to column families that need to replay the operations. For example, for CF_1, transactions before TXN_t can be skipped.

In this way, we can avoid replaying the entire binlog from the beginning and only need to perform partial recovery for each column family as needed. In each column family, the KV items that have been persisted before the local data persistence point are skipped. This not only ensures that we can safely recover all the data but also avoid unnecessary replaying operations, which reduces the involved I/O overhead and accelerates data recovery.

5.5 Reconstructing LSNs

Another challenge in data recovery is how to reconstruct the original logical sequence number (LSN) for each involved KV item. As mentioned in Section 4, each KV item is attached with a unique LSN for version control. After removing the WAL, unfortunately, the LSN information is lost.

If the LSNs were not reproduced correctly, the data would not be recovered with correct version information and stale data could be returned for a query. LSN represents the internal order for KV items in a KV batch (corresponding to a transaction in the RDB layer). The loss of LSNs may lead to erroneous data updates. For example, assuming a KV batch contains two update operations to the same key. In the original system, the two KV items are attached with two unique LSNs. Because LSN is a monotonically increasing number, the KV item with the larger LSN must contain the latest data. However, if we could not recover the LSNs correctly, stale data may be returned, which is unacceptable.

In the current LSM-tree based RDB design, the storage engine layer maintains a global LSN counter, which is incremented by one each time when it is attached to an inserted KV item. Thus, the KV items of a KV batch must have a sequence of consecutive LSNs. As long as we know the original LSN of the first KV item in the batch, we can recover the entire sequence of LSNs of all generated KV items due to the serial property (see Section 5.2). The flush flag contains the last transaction and the LSNs of the first KV and the last persisted KV of the transaction. When replaying a transaction, we simply re-translate the transaction and assign the LSNs one by one. As we know the range of LSNs that are originally assigned, we can derive all the related LSNs for the KV items that need to be recovered.

5.6 Put It All Together

In Figure 4, we show an illustrative example for the replaying process. As shown in the figure, when performing recovery, we first retrieve the flush flags of all the column families. Then we determine the global data persistence point from which we start replaying the transactions and also the local data persistence points from which we perform partial recovery.

In this example, the flush flag of CF_2 contains the local data persistence point TXN_n, the batch's first LSN is $p+1$ and the last persisted LSN is $p+2$. Accordingly, we update the system's current TSN for data recovery to TXN_n, which marks the transaction that we should start replaying from, and set the start LSN to $p+1$. Since the last persisted LSN is $p+2$, the LSN of the next to-be-replayed KV item's LSN should be $p+3$. Then we retrieve the transaction from the binlog and invoke the translation process to regenerate the corresponding KV batch. Note that the first KV item's LSN is $p+1$, but the first KV item that needs be recovered is $p+3$, so we can skip the first two items. During KV replaying, each time when a KV item is allocated with an LSN, we increment the LSN by one, and so on so forth.

6 Evaluation

We have implemented a fully functional prototype of PASV based on Facebook's MyRocks [8], which is a popular LSM-tree based RDB. PASV involves light changes to the existing system (only about 500 lines of C/C++ code), which are mainly in the components for binlog management, persisting data in RocksDB, and transaction-to-KV translation, etc.

In this section, we denote the stock MyRocks as "MyRocks" and our prototype as "PASV". Both MyRocks and PASV use the ROW format in the binlog to log the SQL statements in the MySQL server layer. The WAL of MyRocks is set to the default size. The other parameters for MyRocks and PASV are configured using the default setting from the stock MyRocks. Our experiments are conducted on a workstation equipped with an Intel i7-8700 3.2GHz processor, 32GB memory, and a 500GB SSD. We use Ubuntu 18.04 LTS with Linux Kernel 4.15 and Ext4 file system.

Table 2: Database schema and mapping to column families.

Table Name	Attributes	Format	Primary Key	Secondary Key
	id1	bigint(20)		
	id2	bigint(20)		(id1, link_type,
	link_type	bigint(20)	(link_type, id1, id2)	visibility, time,
LINKTABLE	visibility	tinyint(3)	comment **cf_linkPK;**	id2, version, data)
	data	varchar(255)	**CF_ID=2;**	comment **cf_linkSK;**
	time	bigint(20)		**CF_ID=3;**
	version	int(11)		
	id	bigint(20)		
	link_type	bigint(20)	(id, link_type)	
COUNTTABLE	count	int(10)	comment **cf_countPK;**	
	time	bigint(20)	**CF_ID=4;**	
	version	bigint(20)		
	id	bigint(20)		
	type	int(10)	(id)	
NODETABLE	version	bigint(20)	comment **cf_nodePK;**	
	time	int(10)	**CF_ID=5;**	
	data	mediumtext		

Our workload simulates a typical application scenario supporting Facebook's social network engine, i.e., User Data Base (UDB) [2]. In Facebook, the social graph data includes many object types, such as graph nodes and links, etc. This workload creates a model to simulate the critical pattern of UDB with three major tables, LINKTABLE, NODETABLE and COUNTTABLE, for nodes, relationships and metadata, such as counts of link type for a node, etc. The schema of these three tables are summarized in Table 2. We use LinkBench [1], an open source benchmark tool that simulates UDB-like requests (e.g., SQL INSERT, UPDATE, SELECT, etc.), to generate workloads for evaluating MyRocks and PASV.

Since the data mapping to column families may affect performance, we have extended LinkBench to map the translated KVs into different column families. As shown in Table 2, in the storage engine layer for both MyRocks and PASV, we maintain five column families (CF_ID=1–5). We manually allocate the primary key and secondary key of LINKTABLE, the primary key of COUNTTABLE, and the primary key of NODETABLE to CF2-CF5, respectively. The system column family, cf_system (CF-1), is initialized to store system metadata, such as table schema.

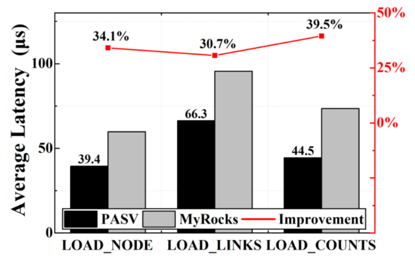
6.1 Overall Performance

We evaluate the performance of PASV and MyRocks for both data loading process and query running process. The data loading process populates the database and only involves write operations. During the query running process, the workloads are mixed with insert, update, and query operations. We use three main metrics reported by LinkBench, namely the total time, throughput, and average latency to compare the performance of PASV and MyRocks.

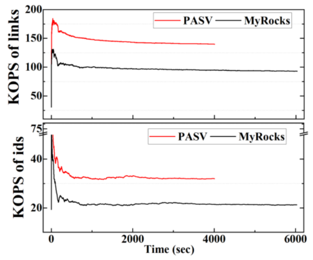
Data loading process. We use 10 loaders to initialize 100 million nodes (*ids*) and their corresponding associations (*links*) in the three tables for both PASV and MyRocks. The

	MyRocks	PASV	Improvement
Total Loading Time (Seconds)	6,027.3	4,019.9	33.3%
Throughput (KOPS)	72.6	108.8	49.9%
Total Binlog Size (GB)	54.4	54.4	
Storage Engine's IO (GB)	206.5	117.9	42.9%

(a.1) General performance (data loading)



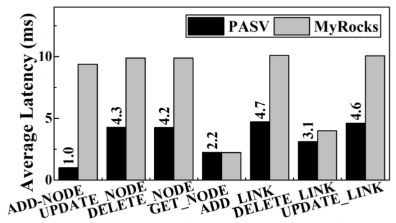
(a.2) Average latency (data loading)



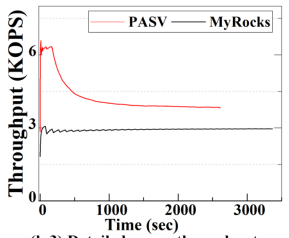
(a.3) Detailed loading throughput

	MyRocks	PASV	Improvement
Total Execution Time (Seconds)	3,371	2,614	22.5%
Throughput (KOPS)	2.97	3.82	28.6%

(b.1) General performance (query running)



(b.2) Average latency (query running)



(b.3) Detailed query throughput

Figure 5: Performance of PASV and MyRocks during (a) data loading process and (b) query running process.

data loading process is write intensive. About 432 million write operations (insert and update) with 100 GB data are issued to the database.

Figure 5(a.1) shows the total loading time and overall throughput. The total loading time of PASV is 4,019.9 seconds, which is 33.3% less than MyRocks (6,027.3 seconds). The throughput for MyRocks and PASV are 72.6 and 108.8 KOPS, respectively, and PASV outperforms MyRocks by 49.9%. Figure 5(a.3) shows the run-time throughput of different phases for PASV and MyRocks. The top sub-figure shows the run-time throughput for loading links, and the bottom sub-figure shows that of loading ids. We can see that since the memory buffers are empty initially, both PASV and MyRocks achieve a high throughput at the beginning (0–200 seconds) of the loading process. After that, the throughput has a sharp decline and finally settles at around 150 and 100 KOPS for loading links with PASV and MyRocks, respectively. For loading ids, the throughput for PASV is around 31 KOPS and 21 KOPS for MyRocks. PASV clearly outperforms MyRocks in both total loading time and throughput. For latency, as shown in Figure 5(a.2), PASV achieves much better performance than MyRocks as well. We divide the system latency into three parts, LOAD_NODE, LOAD_LINKS, and LOAD_COUNTS, as reported by LinkBench. As we can see in the figure, the average latencies of PASV are much lower than MyRocks, which are 34.1%, 30.7%, and 39.5% for the three parts, respectively.

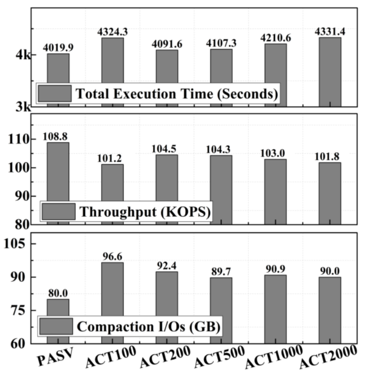
Figure 5(a.1) also shows the I/O cost incurred in the MySQL server layer (caused by writing the *binlog*) and the LSM-tree based storage engine layer. PASV significantly reduces the amount of I/Os in the storage engine layer, compared to MyRocks. This is due to the removal of redundant writes to WAL. Specifically, the total volume of I/Os in the storage engine layer of PASV is 117.9 GB, which is 42.9% less than MyRocks. For the I/Os in the MySQL server layer, both PASV and MyRocks incur 54.4 GB I/Os during the entire loading process, which indicates that PASV does not cause extra overhead for maintaining the binlog.

In summary, by removing WAL in RocksDB, which resides in the critical path, PASV shows strong performance and storage I/O advantages over the stock MyRocks.

Query running process. During the query running process, we invoke 10 clients to perform a mixed set of operations. Each client issues 1 million read and write requests. Note that different from the data loading process, which only involves inserts and updates, the query running process consists of a variety of operations, such as deletes and retrieve queries (e.g., select from primary/secondary key, range scan, etc.) The evaluation results are as follows.

As shown in Figure 5(b.1), the total execution times of the query running process for PASV and MyRocks are 2,614 and 3,371 seconds, respectively, meaning that PASV is 22.5% faster than MyRocks. Figure 5(b.3) further shows the run-time throughput comparison between the two schemes. We can see that during the running process, MyRocks has a small performance fluctuation at the beginning (0–1,000 seconds), and it finally achieves a stable throughput around 3 KOPS. Benefiting from the removal of WAL, PASV significantly outperforms MyRocks from the beginning to 500s, and its throughput gradually settles at around 3.7 KOPS. The overall throughput of PASV is 28.6% higher than MyRocks.

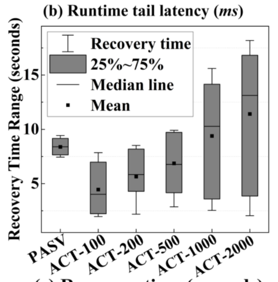
We have also studied the average latencies of different types of requests. As shown in Figure 5(b.2), for the write-intensive requests, such as {ADD, UPDATE, DELETE}_NODE and those for links, PASV outperforms MyRocks across the board. PASV achieves the highest latency reduction in ADD_NODE phase, which is 89.3% lower than MyRocks. Compared to MyRocks, the average latency reduction of PASV for write-intensive requests ranges from 22% to 89.3%. For read-intensive requests (GET_NODE), PASV has slight latency overhead, which is around 0.7% higher than MyRocks. This



(a) Runtime performance and I/Os

	PASV	ACT 100	ACT 200	ACT 500	ACT 1000	ACT 2000
99%	56.9	72.6	62.3	67.2	65.3	68.8
99.9%	121.8	215.2	163.4	160.7	356.7	366.8
99.99%	526.1	632.8	584.3	635.5	827.9	699.4

(b) Runtime tail latency (*ms*)



(c) Recovery time (seconds)

Figure 6: Performance/recovery of PASV and ACTs.

subtle performance difference is mainly due to the relatively lower read hit-ratio of the L0 SSTables in PASV. Since in PASV, the storage engine has no WAL, the I/O pressure is shifted from flush operations to L0 compaction, which makes PASV tend to have more SSTables in L0 than MyRocks. Because the key ranges among the SSTables of L0 may have overlaps, PASV needs to retrieve potentially more SSTables when searching KVs, which takes more time to complete.

6.2 Comparison with Active Flush

PASV adopts a passive approach to flush memory buffers when they reach the size limit (64 MB by default). Active flush, in contrast, synchronizes data persistence by enforcing all memory buffers to periodically flush at the same time, which would incur heavy overhead and weaken the effect of memory buffers. In this section, we compare the two methods.

We implement the active flush method (ACT) based on the `flushAll()` interface in RocksDB, which flushes all memory buffers of the LSM-trees compulsively. We record the transaction commit time during the running process. Once the number of committed transactions reaches a predefined threshold, `flushAll()` is invoked. Since `flushAll()` is a synchronized operation, lock is required during this process and all the commit threads are blocked.

We evaluate ACT by configuring different thresholds and compare their performance with PASV. We have implemented five settings for ACTs. ACT-{100, 200, 500, 1000, 2000} refer to the configurations that invoke the active buffer flush process every corresponding number of transactions, respectively. In order to show the performance during normal runs and data recovery upon failures, a write-intensive workload is created by using 10 loaders issuing 100 million node insertions and corresponding link updates.

Performance. Figure 6(a) shows the overall execution time and throughput of PASV and ACTs (ACT-100 to ACT-2000). We can see that PASV outperforms all the others, taking only 4,019.9 seconds of execution time and achieving a throughput of 108.8 KOPS. Unlike ACTs, which enforce all the LSM-trees' memory buffers to flush at a synchronized time point, PASV does not need to arbitrarily lock and flush the memory buffers periodically. This not only helps retain the efficacy of memory buffers (no premature buffer flushes) but also avoids interfering the foreground requests with unnecessary, costly I/Os. Figure 6(b) shows the effect on the observed tail latencies. The $99th$ to $99.99th$ percentile tail latencies of PASV are much lower than the others, meaning that PASV introduces less interference to foreground requests.

Compaction. Another side effect of active flushes is the more frequently happening compaction operations in the underlying LSM-trees. Figure 6(a) compares the amount of compaction I/Os generated by PASV and ACTs. PASV is much more efficient: The total amount of compaction I/Os is around 80 GB, which is 10.8%–17.2% lower than the ACTs. With frequent flushes, the memory buffers, even not being completely filled up, have to be persisted to storage, which pushes small SSTables into the underlying LSM-trees. As a result, the compaction operations tend to be triggered more often, which in turn amplifies the I/Os.

Data recovery. We have also studied the data recovery performance. We randomly select ten insertion points in the selected time window (1,000–2,000 seconds) to artificially create a simulated "failure", which triggers the recovery process. We collect the time of completing recovery after each failure and show the aggregated results for each configuration in Figure 6(c). As we increase the interval of flushing memory buffers from 100 transactions (ACT-100) to 2,000 transactions (ACT-2000), the recovery time generally increases. This is because the less frequently active flush happens, the more data needs to be restored. PASV achieves a recovery time in the mid-range. Compared to ACTs, PASV shows a much smaller variance in the data recovery time, since our epoch-based approach and partial recovery can identify the last data persistence point and minimize the amount of data for recovery. In contrast, ACTs have to recover data completely since last flush, and the time taken to complete recovery depends on when the failure happens between two consecutive flushes and thus varies significantly.

6.3 Evaluation with TPC-C

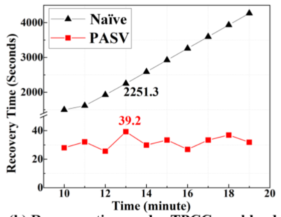
In this section, We perform experimental evaluations using the TPC-C [22] benchmark to compare the performance and potential recovery time of the stock MyRocks, the naïve approach, which simply disables WAL, and PASV.

For better understanding the recovery efficiency of PASV under a more skewed workload, we configure the TPC-C benchmark with 100 warehouses and 10 tables, which are allocated to 10 column families with uneven flush speeds. The data size is around 90 GB in the InnoDB-based MySQL.

Figure 7(a) shows that PASV outperforms MyRocks by 26.4%, 35.9%, and 23.7% in total execution time, throughput (KQPS), and storage I/O amount, respectively. The naïve approach simply disables WAL with no other operations, which represents the possibly achievable performance during normal runs. PASV achieves nearly identical performance to the

	PASV	Naïve	MyRocks
Total Execution Time (Seconds)	2,658.4	2,651	3,612
Throughput (KQPS)	35.6	35.7	26.2
Storage Engine's IO (GB)	204.2	204.2	267.6

(a) General performance under TPCC workload



(b) Recovery time under TPCC workload

Figure 7: Performance/recovery under TPC-C workload.

naïve approach. In fact, PASV only adds a very small KV item (several bytes) during memory buffer flush. The incurred overhead is minimal.

To compare recovery time, we first perform a 10-minute data loading phase for both PASV and the naïve approach, and then simulate a "failure" by artificially turning off the system at different time points (10-19 minutes after the loading phase is completed). As shown in Figure 7(b), although the column families receive KVs at skewed speeds (10x difference), the recovery time of PASV remains low (25-40 seconds), which is far less than that of the naïve approach (at least 1,497 seconds). As the test phase runs longer, the recovery time of the naïve approach increases almost linearly. Compared to PASV, the naïve approach suffers from a time-consuming recovery process, since it has to replay all the transactions in the binlog from the beginning, and the larger the binlog is, the longer it takes for recovery.

6.4 Data Recovery

In this section, we first perform the correctness analysis of the data recovery process in PASV and then we study and compare its recovery performance with MyRocks.

Recovery correctness analysis. By eliminating a redundant logging structure, PASV provides significant performance improvement and still guarantees data persistence, just like the original LSM-tree based RDB. To achieve this goal, upon a system failure, we must ensure that (1) all volatile data stored in the memory buffers before the crash should be recovered completely, and (2) the KVs managed by the storage engine should be consistent with the RDB's view.

The original design provides the above-said data persistence guarantees through a two-phase recovery using both binlog and WAL. In PASV, we must achieve the same goal with binlog only. When a failure happens, PASV first retrieves the local data persistence point (DPP) of each LSM-tree and determines the global DPP from the storage engine layer. Then, it scans all pending transactions from the binlog to determine the KVs that should be replayed. We only need to replay the KVs that have been committed at the RDB layer but not yet been persisted at the storage engine layer.

To analyze the recovery correctness, we divide the data persistence process into three phases based on two key operations `flush(Binlog)` and `flush(MemTable_x)` in Figure 8. The time period $[T_a, T_{a+5}]$ includes all the possible crash points that may happen during the whole process for an LSM-tree.

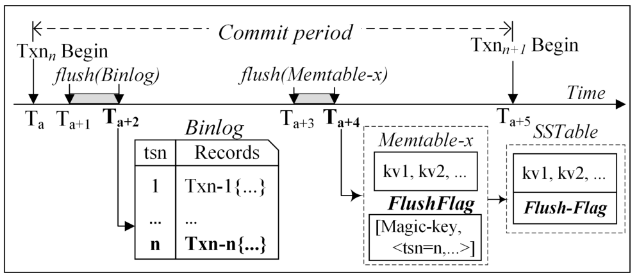


Figure 8: An illustration of transaction commit process.

• *Period #1: Uncommitted to binlog.* During time period $[T_a, T_{a+2})$, the incoming transaction TXN_n is parsed into SQL requests and then being persisted to the binlog. Before T_{a+2}, TXN_n is neither persisted to *binlog* nor the storage engine. If system crash happened during this period, PASV would retrieve the latest flush flag of the LSM-tree from the system. In this situation, the DPP must be a transaction whose sequence number is less than n (TXN_{n-1} or earlier). Thus, during recovery, the partial recovery process would compare current DPP with the global one and only recover the data after the local DPP of this LSM-tree. Since TXN_n is not committed to binlog, it will not be recovered, which is consistent with the status of the RDB layer before the system crash.

• *Period #2: Committed to binlog but not flushed.* During time period $[T_{a+2}, T_{a+4})$, the transaction TXN_n is persisted to the binlog, meaning that all KV items from TXN_n need to be recovered if crash happens. Suppose at time T_{a+3}, the memory buffer of this LSM-tree reaches the size limit, the flush flag (FF_p) containing the current committed transaction TXN_n, the corresponding LSNs and a magic key, needs to be also flushed to disk as the last KV item in the buffer. At time T_{a+4}, the flush operation ends. If a system crash happened during T_{a+2} to T_{a+4}, the flush flag FF_p could be not completely persisted, meaning that we must retrieve the last flush flag FF_{p-1}, which has already been written to the storage in the last round of memory buffer flush. Thus, the last safely persisted transaction should be prior to transaction TXN_n, which is TXN_{n-1} or earlier. Partial recovery can recover starting from there. Since TXN_n is already committed to binlog, all the KVs translated from it will be completely replayed, which is also consistent with the status before system crash.

• *Period #3: Committed and flushed.* After T_{a+4}, the FF_p, which records its current DPP TXN_n and the related LSNs, is persisted to the disk, meaning that for this LSM-tree, the current data persistence status has been persisted to disk. When a failure happens during this phase, PASV will retrieve the latest persisted flush flag, which is FF_p containing TXN_n and the corresponding LSNs, and replay uncompleted KV items from TXN_n after the latest DPP. In this case, the DPP is TXN_n. Partial recovery will only recover KV items of TXN_n that appear after T_{a+4} for this LSM-tree. Hence, it is also consistent with the status before system crash.

In general, PASV maintains at least one local DPP for each LSM-tree indicating the restart point. According to the latest

Table 3: Recovery performance of PASV and MyRocks.

	PASV		MyRocks	
	Average	*Range*	*Average*	*Range*
Recovery Time (Seconds)	7.9	0.11	8.3	0.18
Logging Volumes (GB)	1.3	0.26	3.7	0.48
Recovery I/Os (GB)	0.9	0.04	1.9	0.09

flush flag of each LSM-tree, all possible recovery situations can be solved based on the time periods of the transaction commit when crash happens.

Recovery performance analysis. To evaluate the recovery performance, we create a scenario that both PASV and MyRocks have 0.8 GB data stored in memory buffers. Table 3 shows the evaluation results of repeating the same test for 5 times. For the recovery time, PASV recovers the buffered data within 7.9 seconds on average, which is 4.8% less than MyRocks. Replaying SQL statements incurs more complexities and time cost than replaying KVs directly. When replaying SQL statements, more translation and transaction control are involved, which diminishes the performance gains from less disk I/Os and explains the relatively small improvement.

For space usage, PASV frees disk space needed for WAL, and it only needs to maintain the binlog for crash recovery. The binlog accounts for about 1.3 GB, which allows us to recover all in-memory data in storage engine's buffers and all pending transactions maintained in transaction buffer. In contrast, MyRocks needs to maintain about 3.7 GB data for both binlog and WAL, which is 2.8 times of PASV.

Furthermore, benefiting from the efficient partial recovery, PASV also incurs less disk I/Os (0.9 GB on average) for data recovery, compared to MyRocks (1.9 GB). MyRocks with multiple LSM-trees has to retrieve all the KV items in the WAL to recover each LSM-tree, which leads to severe I/O amplification. Since MyRocks is unaware of the relationship between KV items in WAL and each LSM-tree, for each LSM-tree, MyRocks needs to check all KV items in the WAL and replay the KVs that are related to the LSM-tree. In contrast, benefiting from the epoch-based data persistent policy and partial recovery, PASV can skip the unrelated KV items. Hence, for replaying 0.8 GB data, PASV only involves 0.9 GB disk I/Os, which shows its high efficiency.

7 Related Work

In recent years, storage systems have become increasingly more diverse to meet the new requirements of various emerging applications. A lot of efforts have been particularly made on optimizing data storage performance and reliability.

LSM-tree based RDBs. To satisfy the demand for performance and space efficiency, some RDBs (e.g., distributed RDBs [5], HTAP RDBs [28]) have begun to adopt LSM-tree-based storage engines [5, 27–30, 41]. Specifically, MyRocks [8], Spanner [5, 27], CockroachDB [30] as well as TiDB [28], are relational database systems that support MySQL-style protocol and provide full-featured transactional guarantees. They all adopt LSM-tree-based KV store [12, 42] as storage engines. Our work particularly focuses on solving the double-logging problem in LSM-tree based RDBs.

Journaling of journal. The journaling-of-journal (JoJ) problem is widely existing in modern storage systems, which often results in write amplification and time-costing synchronous I/Os when persisting data to file systems and databases [43–47]. A similar problem is the log-on-log problem [20], which appears when running a log-structured file system on a log-structured flash FTL. These problems are different from the double-logging problem addressed in this paper. For double-logging, the two logs are redundant and stored separately, thus one unnecessary log can be eliminated safely. In contrast, the log-on-log problem happens in a distinct environment where each of the two layers of logs is a must-have needed for different semantics and functionalities, meaning that we cannot easily remove one of them in the way as how we handle the double-logging problem.

NVM-assisted logging. Prior works also propose to adopt byte-addressable Non-volatile Memory (NVM) devices to optimize the logging performance [9, 25, 48–56]. For example, NoveLSM [54] proposes to replace the LSM-tree's memory component with persistent NVM devices. It exploits I/O parallelism by searching multiple levels concurrently to reduce lookup latency. MatrixKV [55] is another hybrid design that combines NVM with DRAM for better performance, in which the WAL is implemented in NVM to prevent data loss from system failures. These prior works cannot handle the double-logging problem. They either conservatively keep redundant logging mechanisms or simply attempt to accelerate logging using a faster device. Our work takes a different strategy and aims to fundamentally address the problem by completely removing the redundant WAL in the storage engine layer.

8 Conclusion

LSM-tree based storage engine is becoming increasingly popular in modern relational databases. A unique and critical issue is the double-logging problem, which incurs high and unnecessary overhead. In this paper, we have systematically studied this challenging problem and proposed a set of mechanisms to optimize the system design for achieving high performance and reliability. Experimental results show that our solution can effectively improve the system performance and accelerate data recovery at low cost.

Acknowledgments

We thank our shepherd, Amy Tai, and the anonymous reviewers for their constructive feedback and insightful comments. This work is supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 15224918), Direct Grant for Research, The Chinese University of Hong Kong (Project No. 4055151), the National Science Foundation for Young Scientists of China (Grant No.61902218), and the National Natural Science Foundation of China (Grant No.92064008).

References

[1] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A Database Benchmark based on the Facebook Social Graph. In *ACM International Conference on Management of Data (SIGMOD)*, 2013.

[2] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[3] MySQL Database Service. `https://www.mysql.com`.

[4] Chin-Hsien Wu, Tei-Wei Kuo, and Li-Ping Chang. An Efficient B-tree Layer Implementation for Flash-memory Storage Systems. *ACM Transactions on Embedded Computing Systems*, 6(3):19, 2007.

[5] David F. Bacon, Nathan Bales, Nicolas Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. Spanner: Becoming a SQL System. In *ACM International Conference on Management of Data (SIGMOD)*, 2017.

[6] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing Space Amplification in RocksDB. In *International Conference on Innovative Data Systems Research (CIDR)*, 2017.

[7] Yoshinori Matsunobu, Siying Dong, and Herman Lee. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.

[8] Yoshinori Matsunobu. InnoDB to MyRocks Migration in Main MySQL Database at Facebook. Technical report, USENIX Association, 2017.

[9] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *ACM International Conference on Management of Data (SIGMOD)*, 2020.

[10] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[11] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[12] RocksDB: A Persistent Key-Value Store for Fast Storage Environments. `https://github.com/facebook/rocksdb`.

[13] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *USENIX Annual Technical Conference (ATC)*, 2017.

[14] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[15] Alex X. Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. Xengine: A Fast and Scalable XACML Policy Evaluation Engine. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2008.

[16] MongoDB: A General Purpose, Document-based, Distributed Database. `https://www.mongodb.com`.

[17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *ACM Conference on Management of Data (SIGMOD)*, 2017.

[18] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[19] Introduction to InnoDB. `https://dev.mysql.com/doc/refman/8.0/en/innodb-introduction.html`.

[20] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't Stack Your Log On My Log. In *Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.

[21] PASV-based MyRocks. `https://github.com/ericaloha/MyRocks-PASV`.

[22] TPC-C Benchmark for MySQL. `https://github.com/Percona-Lab/sysbench-tpcc`.

[23] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-Suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards Building a High-performance, Scale-in Key-value Storage System. In *ACM International Conference on Systems and Storage (SYSTOR)*, 2019.

[24] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[25] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[26] Group Commit for the Binary Log. `https://mariadb.com/kb/en/group-commit-for-the-binary-log`.

[27] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[28] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-based HTAP Database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[29] YugabyteDB. `https://github.com/yugabyte/yugabyte-db`.

[30] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *ACM International Conference on Management of Data (SIGMOD)*, 2020.

[31] KVs Batch in MyRocks. `https://github.com/facebook/rocksdb/wiki/Basic-Operations`.

[32] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[33] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[34] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *ACM International Conference on Management of Data (SIGMOD)*, 2017.

[35] Theo Haerder and Andreas Reuter. Principles of Transaction-oriented Database Recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.

[36] Tianzheng Wang and Ryan Johnson. Scalable Logging through Emerging Non-volatile Memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, 2014.

[37] Keir Fraser. *Practical Lock-freedom*. PhD thesis, University of Cambridge, UK, 2004.

[38] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based Memory Reclamation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[39] Aleksandar Prokopec. Cache-tries: Concurrent Lock-free Hash Tries with Constant-time Operations. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[40] Jeehoon Kang and Jaehwang Jung. A Marriage of Pointer- and Epoch-based Reclamation. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[41] Andrew Pavlo and Matthew Aslett. What's Really New with NewSQL? *SIGMOD Rec.*, 45(2):45–55, 2016.

[42] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[43] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O Stack Optimization for Smartphones. In *USENIX Annual Technical Conference (ATC)*, 2013.

[44] Kai Shen, Stan Park, and Meng Zhu. Journaling of Journal is (Almost) Free. In *USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[45] Daejun Park and Dongkun Shin. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *USENIX Annual Technical Conference (ATC)*, 2017.

[46] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. In *USENIX Annual Technical Conference (ATC)*, 2015.

[47] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[48] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O through Byte-addressable, Persistent Memory. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[49] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. High Performance Database Logging using Storage Class Memory. In *IEEE International Conference on Data Engineering (ICDE)*, 2011.

[50] Shen Gao, Jianliang Xu, Bingsheng He, Byron Choi, and Haibo Hu. PCMLogging: Reducing Transaction Logging Overhead with PCM. In *ACM Conference on Information and Knowledge Management (CIKM)*, 2011.

[51] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[52] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware Logging in Transaction Systems. *Proceedings of the VLDB Endowment*, 8(4):389–400, 2014.

[53] Tianzheng Wang and Ryan Johnson. Scalable Logging through Emerging Non-Volatile Memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, 2014.

[54] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *USENIX Annual Technical Conference (ATC)*, 2018.

[55] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *USENIX Annual Technical Conference (ATC)*, 2020.

[56] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.