DecLog: Decentralized Logging in Non-Volatile Memory for Time Series Database Systems

Bolong Zheng
Huazhong University of Science and Technology

Yongyong Gao
Huazhong University of Science and Technology

Jingyi Wan
Huazhong University of Science and Technology

Lingsen Yan
Huazhong University of Science and Technology

Long Hu
Huazhong University of Science and Technology

Bo Liu
Tongji Hospital, Huazhong University of Science and Technology

Yunjun Gao
Zhejiang University

Xiaofang Zhou
Hong Kong University of Science and Technology

Christian S. Jensen
Aalborg University

ABSTRACT

Growing demands for the efficient processing of extreme-scale time series workloads calls for more capable time series database management systems (TSDBMS). Specifically, to maintain consistency and durability of transaction processing, systems employ write-ahead logging (WAL) that ensures that transactions are committed only after log entries are flushed to disk. However, when faced with massive I/O, this becomes a throughput bottleneck. Recent advances in byte-addressable Non-Volatile Memory (NVM) provide opportunities to improve logging performance by persisting logs to NVM instead. Existing studies typically track complex transaction dependencies and use barrier instructions of NVM to ensure log ordering, and few studies consider the heavy-tailed characteristics of time series workloads, where most transactions are independent of each other.

We propose DecLog, a decentralized NVM-based logging system to enable concurrent logging of TSDBMS transactions. Specifically, we propose data-driven log sequence numbering and relaxed ordering strategies to track transaction dependencies and resolve serialization issues. We also propose a parallel logging method to persist logs to NVM after being compressed and aligned. An experimental study on the YCSB-TS benchmark offers insight into the performance properties of DecLog, showing that it improves throughput by 4.6× at most with less recovery time when compared with the open source TSDBMS Beringei.

PVLDB Reference Format:
Bolong Zheng, Yongyong Gao, Jingyi Wan, Lingsen Yan, Long Hu, Bo Liu, Yunjun Gao, Xiaofang Zhou, and Christian S. Jensen. DecLog: Decentralized Logging in Non-Volatile Memory for Time Series Database Systems. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Chriszblong/DecLog.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

We are witnessing growing demands for time series data management from real-world applications in areas such as cluster monitoring [48], finance [80], and medicine [17], as well as Internet of Things (IoT) [73]. Traditional relational online transaction processing (OLTP) systems have been replaced gradually by time series database management systems (TSDBMS), such as InfluxDB [6], TimescaleDB [11], Monarch [13], Gorilla [58], and Apache IoTDB [66]. In TimescaleDB, transactions that exclusively consist of insert operations are referred to as insert transactions, while transactions that include update or delete operations are referred to as update transactions. To improve throughput and query processing performance, TSDBMSs often write the most recent time series data to DRAM. However, data may be lost when a system failure happens due to the volatile nature of DRAM. To maintain the atomicity and durability transactions, write-ahead logging (WAL) is adopted widely [32]. When a transaction attempts to write data to the database, a corresponding log entry is written first, and the transaction is committed only after its log entries are flushed to non-volatile storage. When data is lost due to a system failure, lost data can be recovered by replaying the log. As WAL is on the critical path of transaction execution, the throughput depends on the logging performance.

Most existing DBMSs adopt the centralized ARIES [50, 51] logging approach to compute a log sequence number (LSN) for each log entry, ensuring that the log entries and the transaction commits are ordered consistently. However, ARIES does not scale well on modern multi threaded CPUs since it relies on a single thread for logging to serve transactions processed in multiple threads, which can cause serious concurrency contention [61, 69, 79]. To improve logging performance, studies propose decentralized logging approaches that persist log records on block storage media, such as Solid State Drives (SSD) and Hard Disk Drives (HDD) [26, 31–33, 61, 72, 79, 81]. However, logging remains a bottleneck of database throughput. We use the Yahoo! Cloud System Benchmark-Time Series (YCSB-TS) to compare the throughput of InfluxDB, TimescaleDB, and Beringei.

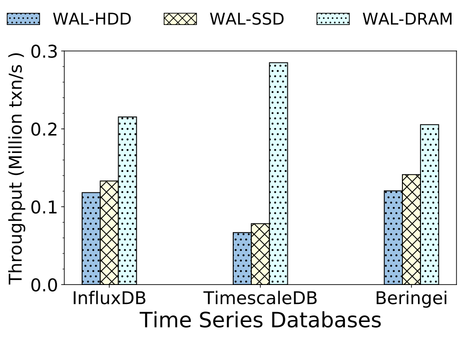


Figure 1: Comparison of TSDBMS

The results are shown in Figure 1, where WAL-HDD and WAL-SSD denote that HDD and SSD are used for logging, respectively, and WAL-DRAM denotes that DRAM is used for logging, which eliminates the I/O cost of logging and yields the best performance possible. We observe that the throughput of WAL-SSD is improved by 15.7% on average compared to that of WAL-HDD, while WAL-DRAM improves performance considerably. The results show that a substantial performance optimization space remains for WAL.

In 2019, Intel's Optane PMem based on 3D XPoint (PMem) [9] was released and became the world's first commercially available byte-addressable non-volatile memory (NVM). Its write latency is comparable to that of DRAM, and the read latency is about 2–3 times that of DRAM. Therefore, recent studies [1, 15, 16, 19, 23, 28, 38, 40, 55, 57, 59, 65, 67, 68, 72] propose to store logs in NVM to improve throughput for OLTP systems. However, since TSDBMS transactions differ considerably from OLTP transactions, the use of NVM for TSDBMS logging still faces the unaddressed challenges:

- **Dependency capture**. Due to the heavy-tailed characteristics of time series workloads, most operations are insertions and queries on the most recent data [45]. Updates and deletes of historical data are rare. Further, the data accessed by update transactions is relatively scattered, with few inter dependencies. How to best design logging to efficiently capture the dependencies among time series transactions is the first challenge.
- **Failure atomicity**. Existing methods rely on *sfence* instructions [4] to ensure correct ordering of log entries, which increases the overhead of logging. However, insert transactions can be executed concurrently without ordering constraints on log entries. Further, although updates in TSDBMSs are rare, update transactions with conflicts must be serializable. How to reduce *sfence* instructions while ensuring correctness is the second challenge.
- **Synchronization**. With more threads available to transactions, centralized logging increasingly causes contention. If we simply adopt parallel logging, additional overhead is incurred by the need for synchronization of multiple logging threads. How to design thread synchronization that enables high logging performance is the third challenge.

We propose DecLog, a decentralized logging system that considers the characteristics of time series data and adopts a three-tier structure of DRAM + NVM + HDD/SSD to persist logs in NVM and the data in HDD/SSD. DecLog features four main innovations: data-driven LSN, a relaxed ordering strategy, parallel logging, and log alignment with compression. In addition, a checkpoint module that does not block insertions is designed that exploits the insert-heavy characteristic of time series data. DecLog also includes a recovery algorithm to achieve fast startup after system failures. In summary, the key contributions are as follows.

- We propose a data-driven LSN approach that exploits the data accessed by transactions to alleviate the computation overhead of the centralized logging and that tracks transaction dependencies based on the characteristics of time series data.
- We propose a relaxed ordering strategy to persist log entries in NVM by using a log flushing pipeline, which can effectively reduce the number of *sfence* instructions, thus reducing the negative impact of log ordering constraints on the concurrency, while guaranteeing atomicity.
- We propose a thread snapshot based parallel logging method to achieve multi-threaded synchronization, the goal being to improve scalability and decrease synchronization overhead.
- We propose a group commit method with log compression and an alignment algorithm to reduce the storage footprint of logs and to fully utilize the write performance of NVM.
- We report on an extensive performance study using the YCSB-TS benchmark, which offers evidence that DecLog is able to improve transaction throughput by 4.6× at most with less recovery time.

The remaining of the paper is organized as follows. Sec. 2 reviews related work. We provide the background and motivation in Sec. 3. Sec. 4 presents the specific design of DecLog. The experimental study is presented in Sec. 5, and Sec. 6 concludes the paper.

2 RELATED WORK

2.1 Time Series Databases

Most TSDBMSs, such as Prometheus [10], Apache Druid [5], and OpenTSDB [3] adopt the widely used Log Structured Merge Tree (LSM-Tree) [54] to ingest massive time series data into permanent storage (HDDs, SSDs). As one of the most popular TSDBMSs, InfluxDB [6] includes a Time Structured Merge Tree (TSM-Tree) to improve insertion performance. In contrast, other TSDBMSs adopt an in-memory storage model to enhance both insertion and query performance. Monarch [13], a globally-distributed TSDBMS, keeps data in DRAM and offers a regionalized architecture to cater for high performance data monitoring and alerting. Scuba [12] automatically expires data (e.g., only maintains data for 1–2 weeks or hours) and keeps the most recent data in-memory. Kdb+ [8] is a column-oriented TSDBMS based on the concept of ordered list, which exploits L1/2 CPU caches according to a small memory footprint (<800kB). Beringei [22] is an open source TSDBMS based on Gorilla [58] that features a delta-of-delta and XOR compression algorithm to process large-scale time series data in DRAM backed by permanent storage for persistence. Due to its low coupling property, the prototype of DecLog is implemented on top of Beringei.

2.2 Write-Ahead Logging

Whenever time series data is inserted into a TSDBMS, the corresponding log entries should be flushed to permanent storage beforehand. This easily makes logging a bottleneck of data insertion. As few studies of WAL in TSDBMS can be found, so we cover studies of WAL in relational and key-value DBMSs, which fall into two main categories.

Centralized Logging. Most systems use centralized ARIES-style [51] WAL for recovery [31]. However, ARIES does not scale well on multi-core processors, as observed in recent studies [31, 61, 69, 79]. Aether [31] proposes a scalable logging approach that utilizes a consolidation array to advance the safe LSN boundary and allow locks to be released earlier, thus reducing lock contention. ELEDA [33] proposes a concurrent data structure called Grasshopper to track on-the-fly logging efficiently. Border-Collie [35] provides a wait-free and read-optimal algorithm for logging, which finds a recoverable logging boundary and completes required operations in a finite number of steps.

Decentralized Logging. Most decentralized logging systems focus on reducing dependencies between log threads to minimize synchronization overhead. Silo [61, 79] implements an epoch-based decentralized logging system that copies transaction-local redo logs to a per-thread log buffer after validation, thus adopting the optimistic concurrency control (OCC). Wang et al. [67] utilize logical clocks [39] to record a Global Sequence Number (GSN) instead of LSN for each transaction, page, and log. Also, a passive batch commit strategy is utilized to realize the synchronization between multi-thread logs. Haubenschild et al. [26] extend GSN based logging and propose a Remote Flush Avoidance (RFA) protocol to reduce the dependency between logs. Taurus [72] features parallel logging and employs an LSN vector to track transaction dependencies. Although these studies realize synchronization effectively, additional protocol and instruction overheads are introduced. In addition, when determining the transaction log order, they do not consider the data accessed by transactions. Unlike these studies, DecLog implements a data-driven LSN method inspired by TicToc [77] to reduce overhead, and it utilizes a synchronization strategy with a logging thread snapshot.

2.3 Non-Volatile Memory and Usage

Studies of systems that utilize NVM can be classified into four categories: NVM B+ tree [14, 20, 30, 44, 47, 56, 64, 76], NVM hash [21, 27, 41, 46, 52, 70, 71, 82, 83], NVM storage engine [15, 34, 38, 40, 42, 60, 62, 74, 78], and NVM logging [16, 19, 23, 28, 37, 59, 67, 70]. Most of the studies focus on reducing the number of *sfence* instructions to improve the write performance on NVM and ensure atomicity. Fang et al. [23] offer an OS calling interface and a log storage manager for NVM to optimize logging performance. Huang et al. [28] maintain a ring buffer in NVM to mitigate performance degradation caused by the space allocation. All log entries for the same transaction are connected by a double linked list so that log entries of the same transaction can be found quickly during recovery. Write-behind logging (WBL) [16] employs a hybrid storage structure with DRAM and NVM that enables transactions to be executed and persisted in NVM directly. After data is written to NVM, the corresponding log entry can be flushed to NVM. So only the timestamp of each committed transaction needs to be recorded in the log entry. Pelly et al. [59] implement a DBMS in NVM to achieve fast failure recovery. They propose a group commit method to reduce the overhead of

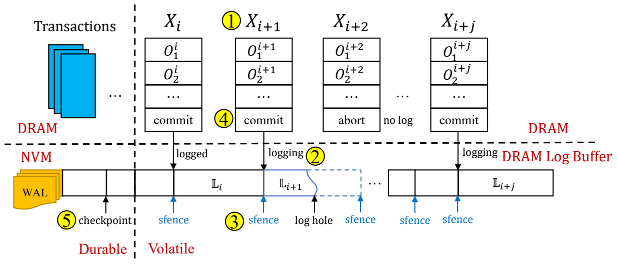


Figure 2: Logging in NVM

sfence instructions, which only needs to persist the undo logs for each group of transactions. DecLog proposes a similar alignment and group commit method, but exploits the characteristics of time series data to improve the logging performance in NVM.

3 BACKGROUND AND MOTIVATION

We proceed to cover the background knowledge and then provide motivation for DecLog.

3.1 Motivating Example

Some recent studies [15, 16, 19, 23, 26, 28, 37, 67] utilize NVM to improve logging performance. To ensure data consistency after recovery, the log order in NVM needs to be consistent with the transaction commitment order. Due to instruction optimization in modern compilers and multi-core CPUs, instruction reordering may cause inconsistencies between different CPU cores. Therefore, when we use the *clwb* instruction to write logs to NVM, the *sfence* instruction [4] must ensure that log entries are persisted in order [29]. In the X86 architecture, the *sfence* instruction sends a cache invalidation signal to the CPU buffer, and flushes the data stored in the buffer to the L1 CPU cache. It also prohibits the CPU from reordering store instructions before and after the *sfence* instruction, which enables all memory updates before the store barrier visible for multi-cores [4]. However, the *sfence* instruction requires a long CPU delay to wait for the *clwb* instruction to complete, which incurs a high execution overhead [24, 75]. The heavy use of *sfence* instructions significantly degrades the logging performance and thus affects the transaction throughput.

Logging in NVM. Fig. 2 illustrates the logging process in NVM. Let X_i denote the i-th transaction, and let O^i denote the operations in X_i, including insert, update, delete, and read. $\mathbb{L}_i$ denotes the log entry of transaction X_i, which has three parts: a header that records meta information, a payload, where the data resides, and a tail that contains the commit flag and LSN. Using transaction X_{i+1} as an example, the logging process is as follows:

① Transaction X_{i+1} is ready to commit.
② An entry $\mathbb{L}_{i+1}$ for X_{i+1} is generated, and its LSN is computed based on the log size $|\mathbb{L}_{i+1}|$ and the LSN of the previous transaction, i.e., $LSN_{i+1} = LSN_i + |\mathbb{L}_{i+1}|$. The current entry $\mathbb{L}_{i+1}$ with LSN_{i+1} is written to NVM.
③ The *sfence* instruction is used to ensure that the head and payload are persisted before the tail, which guarantees the failure atomicity. After $\mathbb{L}_{i+1}$ is persisted, a success flag is returned for transaction X_{i+1}.

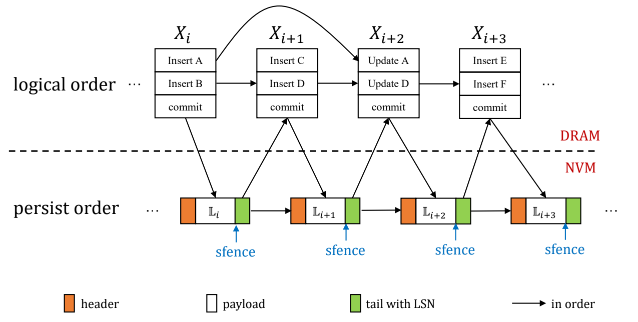


Figure 3: Strict Ordering

④ Transaction X_{i+1} is committed, and its updates become visible to subsequent transactions.

⑤ The checkpoint thread periodically executes in the background, writing dirty pages in DRAM to permanent storage and deleting the corresponding log entries. During recovery, log entries are read from NVM and used for redo operations in order. If a log entry has no log tail, meaning that the transaction has not committed, it is discarded directly.

Strict Ordering Strategy. The above process relies on the *sfence* instruction to ensure a strict ordering of log entries. On the one hand, *sfence* ensures that the head and payload are persisted before the tail of an entry. Without *sfence*, instruction reordering by a compiler and CPU may cause a tail to be persisted before the corresponding head and payload. If a failure occurs at this time, the recovery process incorrectly identifies the transaction as commitment based on the log tail, which leads to an inconsistent recovery. On the other hand, the *sfence* instruction ensures that log entries are persisted in order of their LSNs, in turn ensuring that log entries can be executed in a correct order during recovery.

Fig. 3 illustrates the strict ordering strategy and its negative impact on transaction execution. Four transactions $X_i, \ldots, X_{i+3}$ are executed concurrently in four threads. The *sfence* in log entry $\mathbb{L}_{i+2}$ is necessary because a dependency exists between X_{i+2}, X_i, and X_{i+1}. The concurrency control protocol schedules X_{i+2} to execute after X_i and X_{i+1}, and the *sfence* in $\mathbb{L}_{i+2}$ ensures that $\mathbb{L}_i$ and $\mathbb{L}_{i+1}$ are persisted before $\mathbb{L}_{i+2}$. However, this strict ordering for logging in NVM may result in transaction serialization issues, which reduces transaction throughput. For instance, transactions X_i and X_{i+1} insert four data items at different positions. Before X_i and X_{i+1} commit, they have to wait for entries $\mathbb{L}_i$ and $\mathbb{L}_{i+1}$ to be persisted, where the *sfence* in $\mathbb{L}_i$ causes $\mathbb{L}_i$ and $\mathbb{L}_{i+1}$ to be serialized, so that X_{i+1} has to commit after X_i. However, as no dependencies exist between X_i and X_{i+1}, they can execute concurrently.

In a nutshell, although NVM is able to improve logging performance and enhance transaction throughput, the heavy use of *sfence* instructions caused by the strict ordering strategy leads to significant waiting by the CPU, and thus reduces performance. Further, the transaction serialization caused by centralized LSN computation reduces throughput. In addition, it is difficult to persist log entries with dependencies in parallel due to the thread synchronization overhead.

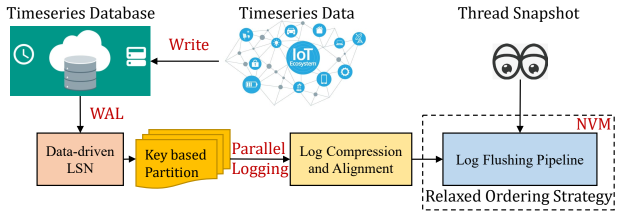


Figure 4: Framework Overview of DecLog

3.2 Framework Overview

To eliminate the limitations mentioned above and to improve logging performance in NVM by utilizing the characteristics of time series data, we propose DecLog, a decentralized logging system, an overview of which is shown in Fig. 4. The system has four key elements: (1) *data-driven LSN* (Sec. 4.1), (2) a *relaxed ordering strategy* (Sec. 4.2), (3) *parallel logging* (Sec. 4.3), and (4) *log compression and alignment* (Sec. 4.4).

When the time series data is inserted or updated in the TSDBMS, corresponding logs are generated, and the LSN of each log is computed by the data-driven LSN module. Next, the parallel logging module assigns the log entries to different log threads according to the key values. After the logs are compressed and aligned by the log compression and alignment module, they are persisted to NVM by using the relaxed ordering strategy. Further, the log flushing pipeline is used to flush logs to NVM concurrently, and thread snapshotting is implemented to achieve synchronization among logging threads.

4 DECENTRALIZED LOGGING

We proceed to introduce the proposed decentralized logging system DecLog, which utilizes both the characteristics of time series and NVM to improve the logging performance in TSDBMS. Due to space limitation, a technical report with more details can be found at https://github.com/Chriszblong/DecLog/DecLog_report.pdf.

4.1 Data-driven LSN

LSN is a unique identifier of each log entry that ensures the order of WALs in permanent storage is consistent with that of the transaction commitments. During recovery, the TSDBMSs replay WAL to restore a consistent state. Most existing studies adopt a centralized logging approach and compute LSN to determine the log order. However, the LSN computation may result in transaction serialization issues, which reduces transaction throughput. Therefore, decentralized logging approaches are proposed to replace centralized ones to improve the logging performance, which are mainly divided into two categories. One is the epoch based logging [18, 61, 79], and the other is the global sequence number (GSN) based logging [26, 67]. Although these studies avoid concurrent contention, additional protocol and instruction overheads are introduced. In addition, they only consider epochs or GSNs, but neglect the data accessed by transactions that can be used to determine whether transactions should be serialized.

Therefore, to effectively evade the transaction serialization issue and improve the concurrent write capability of WAL in TSDBMSs,

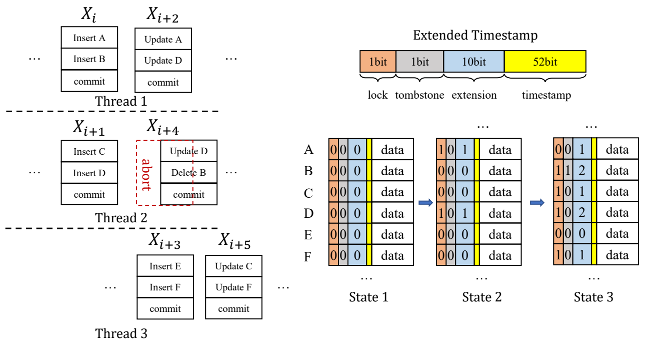


Figure 5: LSN Computing

we propose a data-driven LSN based on the characteristics of time series data. In order to compute LSN, we elaborately design an extended timestamp data structure and an LSN computing protocol.

4.1.1 *Extended Timestamp*.

We first introduce the data structure of extended timestamp. The timestamp field of a time series data point is usually 64 bits, but it suffices to use only 52 bits to represent a timestamp at microsecond-precision, which satisfies the requirements of most real-world applications. Therefore, to fully utilize the timestamp field, we modify its structure to construct an extended timestamp field. As shown in Fig. 5, the extended timestamp is composed of the four parts:

(1) **Lock**. Lock is stored in the highest bit, which is used to achieve the atomic modification of the extended timestamp.
(2) **Tombstone**. Tombstone is stored in the second bit, which is used to indicate whether the data entry has been deleted.
(3) **Update timestamp (*uts*)**. *uts* is stored in the next 10 bits, which is used for LSN computation.
(4) **Timestamp**. The lowest 52 bits are used to store the original timestamp.

We introduce how to set the extended timestamp for a data point in three cases: (1) For a data point newly inserted by a transaction, its extended timestamp is the same as that without using our extended timestamp data structure, since its highest 12 bits are 0s. (2) For a data point read by a transaction, the lock bit needs to be checked to determine whether the data can be read. If the lock bit is 1, the transaction needs to wait until the lock bit becomes 0. Otherwise, the data point can be read directly. (3) For a data point updated or deleted by a transaction, the extended timestamp should be updated with the LSN computing protocol (Details in Sec. 4.1.2).

4.1.2 *LSN Computing Protocol*.

The LSN computing protocol is used to compute LSNs for transactions. We differentiate transactions into three types based on operations: insert-only transactions, read-only transactions, and update transactions. Transactions that exclusively consist of insert operations are referred to as insert-only transactions. Similarly, transactions that exclusively consist of read operations are referred to as read-only transactions. While transactions that include update or delete operations are referred to as update transactions. Let *DP* denote the set of data points accessed by a transaction *X*. Let *dp* be a data point in *DP*, and *dp.uts* be the update timestamp of *dp*. We compute LSN as follows:

Rule 1: The extended timestamp of newly inserted data is 0.
Rule 2: For an insert-only transaction, the LSN of the corresponding log entry is set to 0.
Rule 3: For an update transaction, the LSN of the corresponding log entry is computed by $LSN = \max_{dp \in DP} dp.uts + 1$.
Rule 4: Before a transaction is committed, the *uts* of all modified data points are atomically updated to the LSN of the corresponding log entry.
Rule 5: When a checkpoint is performed, all extended timestamps are set to 0.

Rule 1 ensures that the extended timestamp of the newly inserted data is the same as that without using our design of extended timestamp. Rule 2 ensures that no dependencies exist between insert-only transactions since all the LSNs of these transactions equal to 0. The third and fourth rules ensure that transactions with dependencies are serialized. The last rule avoids the overflow of extended timestamp.

Next, we analyze how to compute LSNs for update transactions as follows: (1) Read extended timestamps and compute LSN. If any lock bit of the extended timestamps is 1, the transaction needs to wait. Otherwise, we compare the *uts* values of all data points in *DP* to compute the maximum *uts*. Then the LSN of the corresponding log entry is computed by $LSN = \max_{dp \in DP} dp.uts + 1$ according to Rule 3; (2) Obtain the lock. We read the lock bits of all modified data and check them one by one. If it is 0, the lock bit is set to 1. Otherwise, the transaction has to be terminated to avoid deadlocks; (3) Modify the extended timestamp. The current *uts* values are compared with those read at the first step. If they are different, it means that another transaction has modified the data, and the current transaction has to be terminated. If all *uts* values are the same, then we modify them to the LSN computed at the first step and set the lock bit to 0. For the deleted data point, the tombstone bit is set to 1.

The LSN computing protocol is similar to OCC, and is suitable for scenarios where transaction conflicts are infrequent. Since there are very few update transactions in TSDBMSs, the LSN computing protocol is able to compute LSNs efficiently while guaranteeing the correctness. We demonstrate the correctness of the LSN computing protocol under two cases. For the first case, when different transactions update the same data, the LSN computing protocol ensures that the LSN is computed according to the order of the transaction commits. For the other case, when different transactions update different data, since there are no dependencies between transactions, redo can be performed in any order to recover the database to a consistent state. Note that when the same data is repeatedly updated by many transactions, the extended timestamp may overflow. To handle the extremely rare situation, we call the checkpoint process to write all dirty data in DRAM to permanent storage, and reset the extended timestamps of all data. The details of the checkpoint are introduced in Sec. 4.5.

We illustrate the LSN computing process in Fig. 5. The worker threads execute transactions $X_i, X_{i+1}, \ldots, X_{i+5}$ concurrently and the logical order is the same as the transaction ID. At first, thread 1 and thread 2 execute transactions X_i and X_{i+1}, respectively. Afterwards, X_{i+2}, X_{i+3}, and X_{i+4} are executed concurrently. For clarity, we assume that X_i, X_{i+1}, and X_{i+3} insert six data points (A to F) into database in sequence and the initial status is shown in state 1. As transaction X_{i+2} updates data A and D, and $A.uts = D.uts = 0$, the LSN is computed by $LSN = max\{A.uts, D.uts\} + 1 = 1$. Then it sets the lock bit to 1, and updates $A.uts = D.uts = 1$ as shown in state 2. Meanwhile, transaction X_{i+4} that runs concurrently with X_{i+2} in thread 2 has to abort because it finds that the lock bit of data point D has been set to 1. Once X_{i+2} commits, it resets the lock bit of data A and D. Next, the database schedules thread 2 to re-execute transaction X_{i+4} and thread 3 to execute transaction X_{i+5} concurrently. At this time, X_{i+4} finds that $B.uts = 0$ and $D.uts = 1$. So, the LSN of its transaction log entry is computed by $LSN = max\{B.uts, D.uts\} + 1 = 2$ according to the protocol and then $B.uts = 2$ and $D.uts = 2$ as shown in state 3. The tombstone bit of data point B is set to 1, which indicates that this data point has been deleted. For transaction X_{i+5}, it computes $LSN = 1$ in the same way as transaction X_{i+2}. Note that our LSN computing protocol allows log entries to have the same LSNs. Although the LSNs of the two log entries for transactions X_{i+2} and X_{i+5} are the same, the database can still be recovered to a consistent state by replaying the log entries in the order of LSN. This is because these two transactions update different data. In addition, although transaction X_{i+4} has dependencies with the previous committed transaction X_{i+2}, the LSN computing protocol ensures that the LSN of X_{i+4} is greater than that of X_{i+2}. Therefore, the recovery in ascending order of LSN can also ensure the correctness.

4.2 Relaxed Ordering Strategy

We proceed to discuss how to persist log entries in NVM efficiently. In relational and key-value databases, the *sfence* instructions ensure that the log entries and the transaction commits are ordered consistently for update-heavy transactions. However, a set of *clwb* and *sfence* instructions adds about 250ns latency on average [53], compared with 90ns latency on average for NVM writes [43]. In fact, transactions in TSDBMSs are insert-heavy, the *sfence* instruction is not needed between log entries of two insert transactions while guaranteeing the atomicity and correctness.

To reduce the number of *sfence* instructions, a straightforward method is to remove *sfence* instructions from log entries of insert transactions, and let CPU schedule the persist order. Fig. 6 illustrates the straightforward method through an example. We only keep the *sfence* instruction in the log entry of update transaction X_{i+2}. So, the log entries $\mathbb{L}_i$, $\mathbb{L}_{i+1}$, and $\mathbb{L}_{i+2}$ are written to NVM concurrently, which avoids the serialization of transactions X_i and X_{i+1}. Note that the *sfence* instruction requires $\mathbb{L}_{i+3}$ to be persisted after $\mathbb{L}_{i+2}$, which results in transaction X_{i+3} to be blocked until X_{i+2} is committed. However, there is no dependency between transactions X_{i+2} and X_{i+3}, and these two transactions could have been executed concurrently. From the above example, we can see that the straightforward method only solves the serialization issue between transactions X_i and X_{i+1} caused by the strict ordering strategy, but is unable to address the serialization issue between transactions X_{i+2} and X_{i+3}.

Log Queues. To solve the above problem, we differentiate log entries by LSN and buffer them in different log queues. Specifically, we logically maintain n log queues q_i $(i \in [1, n])$ with size of l, which are initialized to be empty. When a transaction executes,

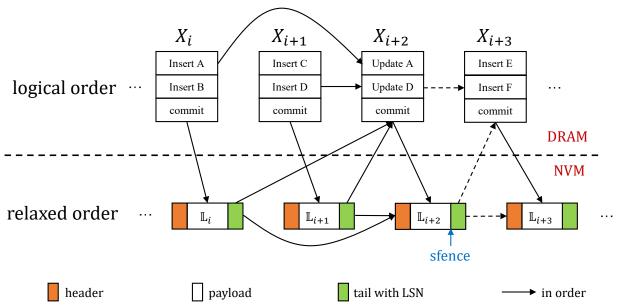


Figure 6: Relaxed Ordering

we first check the *uts* of the accessed data and then compute the LSN of the log entry by data-driven LSN. Once the transaction is committed, the transaction type can be determined by LSN. If $LSN = 0$, then it is an insert-only transaction, the log entry is buffered in q_i. Otherwise, we buffer the log entry in q_i and close the current log queue q_i, so that subsequent log entries are written to the next log queue q_{i+1}. Log entries are buffered to n log queues in a loop manner, i.e., when q_n is closed, the subsequent log entries are buffered to q_1. The n log queues form a circular linked list-like structure in logic.

Log Flushing Pipeline. Next, we discuss how to persist log entries in log queues to NVM. The straightforward approach is to flush log entries based on the log queue ID, but this approach also results in serialization due to the *sfence* instructions. To alleviate the performance degradation caused by *sfence* and maximize the byte-addressable capability of NVM, we propose a log flushing pipeline to persist log entries concurrently, which reduces the waiting overhead caused by the *sfence* instructions.

We illustrate how the log flushing pipeline works with a running example. As shown in Fig. 7, each log entry in Fig. 5 is generated in the order in which the corresponding transaction is executed. Log entries $\mathbb{L}_i$ and $\mathbb{L}_{i+1}$ with $LSN = 0$ are first buffered in q_1. Once $\mathbb{L}_{i+2}$ with $LSN = 1$ arrives in q_1, we temporarily close q_1. The following entries $\mathbb{L}_{i+3}$ and $\mathbb{L}_{i+4}$ are buffered in q_2. As the LSN of $\mathbb{L}_{i+4}$ is 2, we temporarily close q_2 and buffer $\mathbb{L}_{i+5}$ in q_3. The closed queues will be reopened after the log entries in each queue have been persisted in NVM. Specifically, we start from q_1 and use *clwb* instruction to persist $\mathbb{L}_i$, $\mathbb{L}_{i+1}$, and the head and payload of $\mathbb{L}_{i+2}$. Note that transaction X_2 can not commit until the tail of $\mathbb{L}_2$ is persisted. Once the *sfence* instruction in q_1 is executed, the logging thread concurrently flushes $\mathbb{L}_{i+3}$ and the head and payload of $\mathbb{L}_{i+4}$ in q_2, while waiting for the tail of $\mathbb{L}_{i+3}$ in q_1 to be persisted. After the *sfence* instruction in q_1 completes, the tail of $\mathbb{L}_{i+2}$ in q_1 can be persisted in NVM and the transaction X_2 commits. In this way, X_{i+3} can commit before X_{i+2} commits , which avoids the unnecessary dependency of them as shown in Fig. 6. Meanwhile, the *sfence* instruction in q_2 ensures that the last log entry $\mathbb{L}_{i+2}$ in q_1 is persisted before the last log $\mathbb{L}_{i+4}$ in q_2 persisted, which guarantees that the order of log entries persisted in NVM are the same as the logical order of transactions. The following log queues flush log entries in the same manner, which composes a log flushing pipeline.

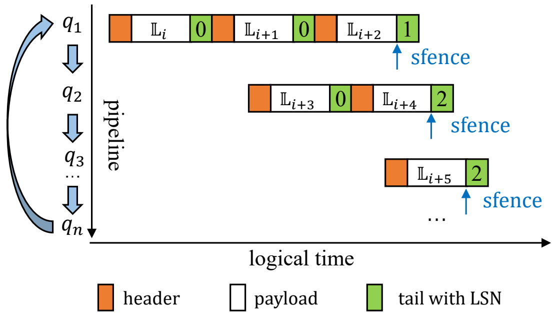


Figure 7: Log Flushing Pipeline

4.3 Parallel logging

We proceed to introduce the thread snapshot based parallel logging method, which is able to further improve logging performance with less multi-thread synchronization overhead.

The parallel logging has been proposed in recent studies [26, 36, 38, 67, 72, 79]. However, there are few commercial databases that distribute log entries in parallel threads within a single node, and even distributed commercial databases still opt for a shared log [31]. Because dependencies among transactions are so widespread and frequent in OLTP systems, it is almost infeasible to track them. If the parallel logging is implemented in OLTP systems, most transactions need to flush multiple log entries at commit time, and the synchronization among logging threads results in performance degradation. However, dependencies among transactions in TSDBMS are so rare and the data-driven LSN can capture the dependencies efficiently, which makes it possible to implement parallel logging.

4.3.1 ***Key-based Partitioning.*** Existing parallel logging schemes are mainly divided into two types: transaction-based partitioning and page-based partitioning, which are designed for the performance optimization of OLTP systems. However, they are not appropriate for TSDBMSs. On the one hand, the vast majority of transactions in TSDBMSs are insert-only transactions, and our data-driven LSN is able to make these transactions independent. So there is no need to apply transaction-based partitioning for thread synchronization. On the other hand, time series data points with the same keys are usually stored continuously. Since the collection periods of data points with different keys are not always the same, the storage footprints of them are different. Therefore, these data points may cover multiple pages or be stored in the same page with others. It is difficult for page-based partitioning scheme to efficiently handle both scenarios.

To overcome the above shortcomings, we implement a key-based partitioning method [58], which records log entry for each insert or update operation in a transaction. Specifically, for each time series data point accessed by a transaction, we use the measurement, tag key, and field key as the key, which is recorded in each log entry. Further, we use hash mapping to map the log entries in a transactions based on their keys to disparate logging threads. However, the key-based partitioning scheme still has two problems: 1) Redo needs to be executed based on transactions across multiple log files during the recovery process, which requires time-consuming log parsing; 2) The log entries for the same transaction need to be recorded in multiple log files and persisted simultaneously, which results in additional thread synchronization overhead. For the first problem, since the dependency of transactions can be tracked by data-driven LSN, we can redo logs by the granularity of operation in the order of LSN, rather than by the granularity of transaction. The details of recovery will be introduced in Sec. 4.6. For the second problem, we propose a thread snapshot to alleviate the synchronization overhead.

4.3.2 ***Thread Snapshot.*** When log entries of an update transaction is to be persisted in NVM, we obtain the LSN of all log entries buffered in each log queue, which is called a thread snapshot. Then we compare the LSN of the current processed log entry with the thread snapshot. If the former is greater, it means that the log entries that have dependencies with the current log entry have not been persisted. So the current log entry needs to wait. Otherwise, we compare the number of persisted log entries with $LSN > 0$. If the number is consistent in the snapshot, the current log entry has to wait. Otherwise, the current log entry can be directly persisted in NVM. Different from latch based synchronization methods [49], the thread snapshot based method is latch-free and enables efficient persistence of log entries for update transactions without blocking independent log entries in other log queues.

Specifically, given m logging threads, we maintain n log queues for each thread. To obtain the thread snapshot efficiently, we maintain a log number array $\mathcal{N}$ with size of $m \times n$ in DRAM, which is used to record the number of update transaction log entries in each log queue that have already been persisted in NVM. We also maintain an LSN array $\mathcal{S}$ with size of $m \times n$ in DRAM, which is used to record the LSN of the log entries in each log queue that have not been persisted in NVM. Both of them are initialized as $\mathcal{N} = \mathcal{S} = [0, \ldots, 0]$. We note that when insert-only transaction log entries are processed, $\mathcal{N}$ and $\mathcal{S}$ do not have to update. When a log entry with $LSN > 0$ is buffered in a log queue $q_{ij}, i \in [1, m], j \in [1, n]$, we record $\mathcal{S}_{ij} = LSN$. Once the log entry is persisted in NVM, $\mathcal{N}_{ij}$ is incremented by 1. Then we update $\mathcal{S}_{ij} = 0$, because each log queue only has one log entry with $LSN > 0$ according to the relaxed ordering strategy.

Next, we illustrate how to use the thread snapshot to handle synchronization for parallel logging with a running example. As shown in Fig. 8, three logging threads, each with two log queues, are used to process log entries for transactions in Fig. 5. Solid-lines indicate that log entries have been persisted in NVM, dotted-lines represent the entries are still being processed in buffer, and gray means that the log entry has not been generated yet. For clarity, we show the data recorded in each log entry by a superscript. We map the log entries by our key-based partitioning. So that data points A and B are mapped to logging thread 1, C and D to logging thread 2, and E and F to logging thread 3. Consider the case where $\mathbb{L}^{B}_{i+4}$ in log queue q_{12} is to persist in NVM, and its LSN is 2. We temporarily neglect the log entry $\mathbb{L}^{c}_{i+5}$ in q_{21} for now. At first, logging thread 1 reads the current thread snapshot, which is shown as state 1. Since the log entry $\mathbb{L}^{A}_{i+2}$ has been persisted in NVM, we have $\mathcal{N}_{11} = 1$. As the LSNs of $\mathbb{L}^{B}_{i+4}$, $\mathbb{L}^{D}_{i+2}$, and $\mathbb{L}^{D}_{i+4}$ are 2, 1, and 2, respectively, we have $\mathcal{S}_{12} = \mathcal{S}_{22} = 2 > \mathcal{S}_{21} = 1$. Therefore, we know that the log entries in other log queues whose LSN is smaller than that of $\mathbb{L}^{B}_{i+4}$ have not been persisted in NVM. The logging thread 1 needs to wait and

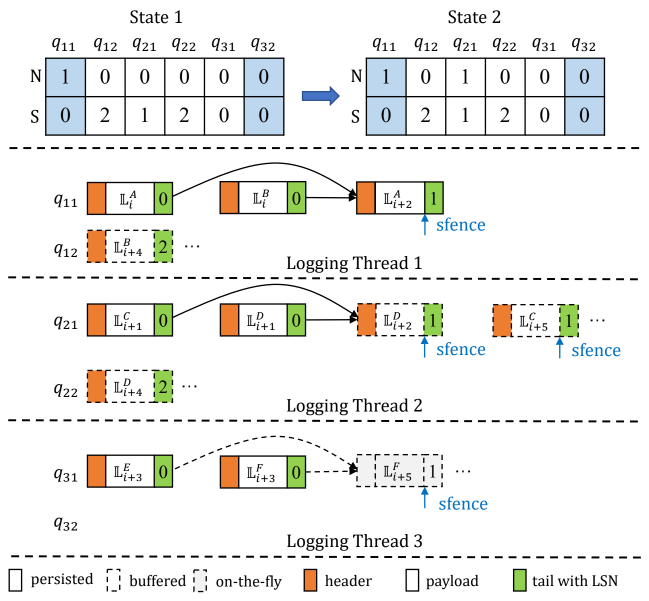


Figure 8: Snapshot

reads the thread snapshot in loop. After $\mathbb{L}_{i+2}^D$ is persisted in NVM, the log entry $\mathbb{L}_{i+5}^C$ is buffered in q_{21} exactly. The logging thread 3 reads the updated thread snapshot shown as state 2. Although $\mathcal{S}_{21}$ is smaller than $\mathcal{S}_{12}$ at this time, the former log entry $\mathbb{L}_{i+2}^D$ has been persisted in NVM and the data-driven LSN protocol ensures that $\mathbb{L}_{i+5}^C$ in q_{21} has no dependency with $\mathbb{L}_{i+4}^B$ in q_{12}. Therefore, $\mathbb{L}_{i+4}^B$ can be persisted in NVM safely.

4.4 Log Compression and Alignment

We proceed to introduce the log compression algorithm and log alignment based group commit method to further improve the logging performance.

4.4.1 ***Compression.*** With the large volume of time series data continuously inserted to TSDBMS, the corresponding log entries need to be persisted in NVM, which requires considerable storage footprint. In addition, the write endurance of Optane PMem is eight orders of magnitude lower than DRAM [43]. If such a large scale of log entries are persisted in NVM, the lifetime of NVM will be shortened. Therefore, log compression is crucial for TSDBMS to reduce the storage footprint and extend the lifetime of NVM. We observe that data in each log entry exhibits the following characteristics:

(1) The metadata that includes page ID, segment position, LSN, etc., are all integers. Only the key of time series data is string.
(2) The number of keys is relatively small and stable in TSDBMS.
(3) For time series data with the same key, the collection interval is usually fixed, and the range of value fluctuations is relatively small.

By exploiting the characteristics of time series data, we adopt the delta-of-delta and XOR compression algorithms to compress the numerical values [58]. These two compression methods achieve considerable performance for continuously varying data. Therefore, it can be rightly applied for time series data with the same key. However, log entries are persisted in the LSN order and not classified by keys. So it is challenging to achieve the best performance of the two compression algorithms. To compress the time series data in logs according to its key, we construct a hash table for keys in NVM and maintain a mirror of the hash table in DRAM, which is used to classify and compress data based on their keys. The key of the hash table is the key of the time series data, and the values of the hash table contain compressed string encoding, integer field, floating-point number field, floating-point number encoding, and log metadata which belongs to the first log entry with this key. The integer field is used for delta-of-delta timestamp compression and the two floating-point number fields are for the XOR values of floating point number. Except for the string encoding, all other fields are cleared after each checkpoint. The hash table is first created in NVM and then synchronized to the DRAM mirror. So that the reading of the hash table during the log compression is completed entirely in DRAM. For each log entry $\mathbb{L}_i$, a 1-bit compression flag is added to indicate whether the entry is compressed or not. Before the log entry is persisted, we first check whether the key of the data exists in the hash table. If it exists, we read the original data from the hash table, compress the entry, and set the compression flag to 1. Otherwise, we first set the compression flag of the entry to 0, and persist the current log entry in NVM. Then we udpate the corresponding entry in the hash table, persist it in NVM, and synchronize the DRAM mirror.

4.4.2 ***Group Commit with Log Alignment.*** The Optane PMem hardware works internally on 256-byte blocks, while the typical CPU cache line is 64 bytes. So persisting log entries to NVM in 64 bytes causes about three times the write amplification. Besides, the write performance of NVM in 64 bytes is about three times worse than that in 256 bytes [63]. Hence, we align the log entries in 256 bytes after they are compressed to maximize the performance of NVM with slightly more space overhead. Specifically, for each log queue in a logging thread, we merge all log entries whose length are smaller than 256 bytes into it. Until the total data size approaches 256 bytes or its multiple, we pad the log entry with 0s at the end to align it to 256 bytes. After it, we persist the log entries to NVM, and commit the corresponding transactions in group [59]. With 256 bytes alignment, the starting address of the log entry that is larger than 256 bytes is a multiple of 256 bytes, which avoids to use LSN to compute the offset of each log entry in NVM. Note that with the alignment, the size of aligned log entries may not be equal to 256 bytes or its multiple in most cases, which results in a slight storage footprint waste. However, without alignment, a log entry smaller than 256 bytes will be persisted to NVM as complete 256-byte data, which results in write amplification and performance degradation.

4.5 Checkpoint

With the surge of data in TSDBMS, the storage footprint of logs continuously increases, which leads to longer recovery times. Besides, the LSN computing protocol may cause LSN overflow when the same data point is updated intensively, which is rare in TSDBMS but still needs to be addressed. Therefore, we periodically execute a checkpoint process to write the dirty data in DRAM to permanent storage, and purge outdated logs. In addition, we need to record the checkpoint log and reset the extended timestamp. However, these

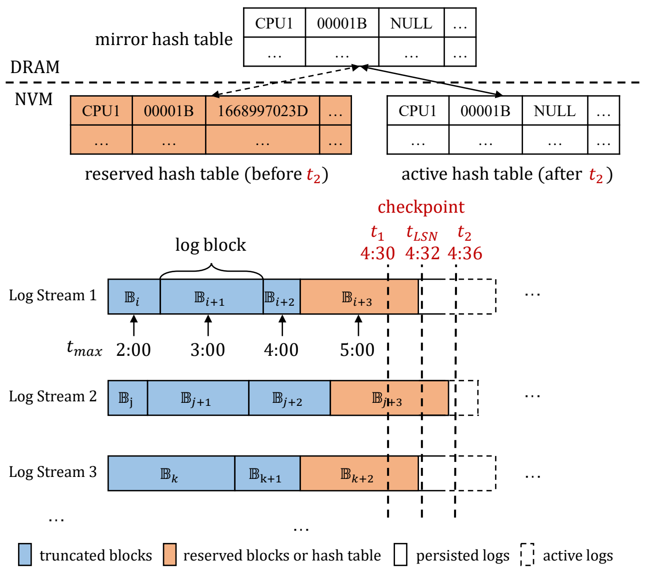


Figure 9: Checkpoint

operations have to block transactions, which affects the performance of TSDBMS. To tackle the problem, we design a checkpoint process based on DecLog, which does not block insert transactions to alleviate the performance degradation. The main difference from the routine checkpoint is the process to record checkpoint log.

Checkpoint Log. The checkpoint log comprises of four parts: (1) The time period $[t_1, t_2]$, where t_1 and t_2 denote the start and end time of checkpoint, respectively; (2) The offsets of log blocks within $[t_1, t_2]$; (3) The LSN update time t_{LSN}; (4) The hash table. The creation of the checkpoint log includes the following four steps: (1) After blocking update transactions, we write dirty data to permanent storage, and record the start time t_1 of the checkpoint; (2) Outdated logs are purged and log entries in each log queue are persisted in NVM. The offsets of current processing log blocks are recorded in the checkpoint log; (3) The extended timestamp of all updated data in DRAM is set to 0. The insert-only transactions can continue to be executed while update transactions must be blocked at this point. The completion time t_{LSN} of this step is recorded in the checkpoint log; (4) A copy of the active hash table in DRAM is persisted in NVM for recovery. The active hash table is reset except for the string encoding of the key, and all other fields are cleared. We record the end time t_2 at this step. After the checkpoint log is persisted, both of insert and update transactions can be executed and the new hash table is used to compress logs from time t_2.

As shown in Fig. 9, we illustrate the checkpoint process with a running example. (1) The checkpoint starts at time $t_1 = 4 : 30$ which is persisted to NVM as part of the checkpoint log, and all time series data prior to t_1 is also flushed to permanent storage. (2) Outdated logs are purged. Taking the log stream 1 as an example, since the maximum timestamp of the data in log blocks $\mathbb{B}_i$, $\mathbb{B}_{i+1}$, and $\mathbb{B}_{i+2}$ is less than t_1, these log blocks are directly purged. As the maximum timestamp of the log entries in the log block $\mathbb{B}_{i+3}$ is greater than t_1, $\mathbb{B}_{i+3}$ needs to be preserved as part of the checkpoint log. Next, all the log entries in log queues are persisted in NVM and the offset in $\mathbb{B}_{i+3}$ is recorded in the checkpoint log. (3) All extended timestamps are set to 0, and the completion time $t_{LSN} = 4 : 32$ is recorded. (4) The active hash table is copied in NVM firstly and all fields are cleared except for the string encoding afterwards. When the end time $t_2 = 4 : 36$ is recorded in the checkpoint log, we synchronize the mirror hash table with the active one in NVM and the checkpoint finishes.

4.6 Recovery

DecLog takes the no force/no steal policy and records redo logs which are compressed and aligned, so we divide the recovery for DecLog into two steps: 1) Log parsing. Log entries are read from NVM and decompressed. 2) Log replay. The most recently committed changes are recovered by the information stored in logs. As logs of insert-only transactions can be replayed concurrently, we first replay these logs in parallel during log parsing, then update transaction logs are replayed in the order of LSN. Next, we introduce the main modifications made to the existing recovery method in our recovery algorithm of DecLog as follows:

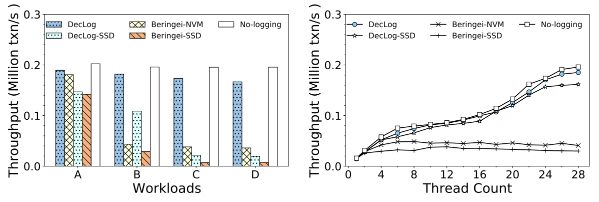
Log Parsing. If there is no checkpoint log, log parsing process starts from the first log block, and the active hash table in NVM is used to decompress logs directly. Otherwise, we obtain the offsets of log blocks from the checkpoint log and start from the offsets to read logs. Next, we introduce how to parse logs and simultaneously replay insert transaction logs.

We create one thread for log parsing and multiple threads to replay logs of insert transactions. To buffer the log entries of update transactions based on LSN, we create a set of buckets whose number is equal to the maximum LSN. In addition, the hash table in the checkpoint is used to decompress log entries with data timestamps before t_2, and the active hash table is used to decompress log entries with data timestamps after t_2. Specifically, let $addr_i$ denote the offset of a log block recorded in the checkpoint, and $addr_j$ denote the current address of the log entry that is read by log parsing thread. At initialization, $addr_j$ is equal to $addr_i$. The log parsing thread reads log entries from $addr_j$ to $addr_j + 256$ and decompresses them by the hash table. Meanwhile, it sends log entries with $LSN = 0$ to log replay threads, and log entries with $LSN > 0$ are buffered in the corresponding bucket. If log parsing thread encounters an end flag, $addr_j$ is updated by $addr_j = addr_j + 256$. Otherwise, log parsing thread continues to read n consecutive 256 bytes until the end flag is encountered and $addr_j$ is updated by $addr_j = addr_j + n \times 256$. This process continues until l consecutive data blocks of 256 bytes are all 0s, where l is the length of log queue. As transactions are committed in group. If log entries parsed from a 256-byte aligned block are incomplete, none of them needs to be replayed.

Log Replay. We introduce how to process update transaction logs in this step. Since the data-driven LSN captures the dependencies between transactions and computes LSN monotonously, update transaction log entries with the same LSN indicate that there are no dependencies between these transactions. Therefore, we can replay update transaction logs in the order of LSN and the correctness is guaranteed. Specifically, we divide the update transaction logs in the buckets into two parts, with one part before t_{LSN} and the other part after t_{LSN}. Then we create a single thread to replay the update transaction logs of the two parts in the order of LSN respectively.

Table 1: Workloads Profile

Parametres	Workloads			
	A	B	C	D
insert	100%	90%	90%	90%
read	0%	9.9%	9%	9%
update	0%	0.1%	1%	1%
θ of zipfian	-	0.9	0.9	1.1



(a) Throughput Comparison (b) Scalability

Figure 10: Throughput Comparison of Logging Strategies

5 EVALUATION

We report on extensive experiments on the benchmark YCSB-TS that offer insight into the performance of DecLog.

5.1 Experimental Settings

We implement DecLog on top of the open source TSDBMS Beringei [22]. Due to the lack of support for update and delete operations in Beringei, we modify the source codes to implement these functions. DecLog is implemented in C/C++ and compiled by GCC/G++ 5.5.0 with -O2. We conduct all experiments on a two-socket server running Ubuntu 18.04 LTS with two Intel Xeon Gold 6326 @ 2.9 GHz CPUs, each of which has 16 cores and 32 threads. To avoid the CPU context switching overhead, we close CPU hyper-threading and each thread is bound to its corresponding CPU core via affinity. Each CPU is equipped with 4 channels of Intel Optane DC Pmem DIMMs 200 Series (128 GB per channel, 1 TB total) and 4 channels of DDR4 memory (3200 MHz, 128 GB total). The hard disk comprises of 4 Nand Flash SSDs (4 TB total) and 3 HDDs (7.2 TB total). In addition, we use the libpmem of Intel Persistent Memory Development Kit (PMDK) library [7] to manage NVM.

Competitors. We implement and compare five logging methods: (1) **DecLog**: our proposal in NVM; (2) **DecLog-SSD**: the logs of DecLog are persisted in SSD; (3) **Beringei-NVM**: the logs of Beringei are persisted in NVM by a strict ordering strategy of centralized logging; (4) **Beringei-SSD**: the logs of Beringei are persisted in SSD; (5) **No-logging**: the logging module of Beringei is disabled.

Parameters. The number of logging threads in DecLog is set to 4. Each thread is dedicated to a single channel of NVM, and each thread is composed of 2 log queues with log queue length of 300.

Workloads. We use the time series benchmark YCSB-TS [2] to test each performance metric of all proposals. To evaluate the performance of update transactions, we modify the Core Workload of YCSB-TS to generate workloads in accordance with a given ratio for insert, read, and update transactions. Then we vary the portion of transactions and follow the widely-used Zipf distribution [25] to generate four workloads (workload A–D). Each transaction in the workloads has 10 operations. The parameter settings for the four workloads are shown in Table 1, where θ represents the parameter of the Zipf distribution. Compared with workload C, there are more transactions that access the same data in workload D. So that workload D presents a higher level of transaction conflict. Each data point has a total size of 50 bytes, which comprises an 8-byte timestamp, an 8-byte data field, and a 34-byte character field. The total size of the data is 18GB, which consists of approximately 400M individual data points.

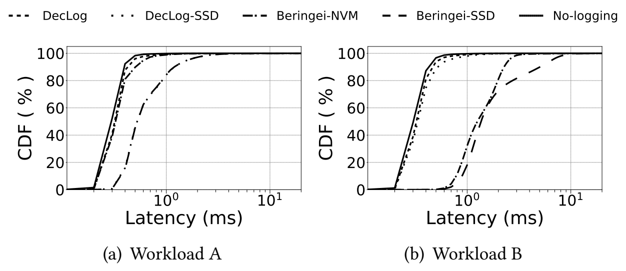
5.2 Overall Performance

We conduct a comprehensive performance study for our proposals in terms of transaction throughput, scalability, commit latency, and recovery, respectively.

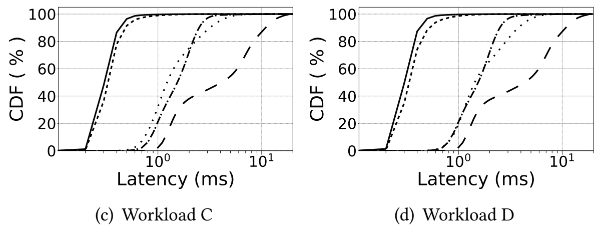
Transaction Throughput. We compare the transaction throughput of five approaches on all workloads. As shown in Fig. 10(a), we make the following observations: (1) For the workload A–D, the transaction throughput of DecLog is 1.1×, 4.2×, 4.5×, and 4.6× that of Beringei-NVM, respectively. Compared with Beringei-SSD, the transaction throughput of DecLog-SSD is improved by 1.1×, 3.8×, 3.1×, and 2.7× on the four workloads, respectively. The reason is that Beringei serializes transactions because of the strict ordering strategy and centralized LSN computation for each log. Conversely, DecLog and DecLog-SSD compute the LSN based on the accessed data and persist log entries in a pipeline, which allows for parallel logging and the concurrent execution of independent transactions; (2) Compared to DecLog-SSD, the throughput improvement of DecLog is 1.1×, 1.7×, 8.0×, and 8.5×, respectively. The throughput gain is beneficial from the low write latency and high bandwidth of NVM; (3) For all workloads, the transaction throughput of DecLog is comparable to that of No-logging and is decreased by 6.2%, 7.5%, 11.3%, and 15.6%, respectively, which demonstrates the effectiveness of DecLog for optimizing logging performance; (4) Compared with workload A, the transaction throughput of DecLog is decreased by 8.2% and 12.7% on workload C and workload D, respectively. This is mainly due to the increased overhead of duplicate LSN computations as the conflicts among transactions increase. However, since update transactions in TSDBMSs are typically infrequent, the effect on database performance is negligible.

Scalability. To evaluate the scalability of our proposals, we vary the number of transaction threads from 1 to 28 and test the corresponding throughput. The results of five approaches are shown in 10(b). We can see that as the number of threads increases, DecLog and DecLog-SSD exhibit approximately linear throughput growth, which shows good scalability. This is because that DecLog applies data-driven LSN and thread snapshot based parallel logging, which reduces the log contention overhead resulting from an increase in the number of transaction threads.

Commit Latency. To evaluate the commit latency of all proposals, we rank the transactions in an ascending order of their commit latency, and compute the average commit latency of the top 95% transactions. Fig. 11 shows the cumulative distribution function (CDF) of average commit latency. For the workload A–D, the average latency of DecLog-SSD is shorter than Beringei-SSD by up to



(a) Workload A (b) Workload B



(c) Workload C (d) Workload D

Figure 11: CDF of Transaction Latency

16.4%, 83.5%, 68.4%, and 62.1%, respectively. Compared to Beringei-NVM, the average commit latency of DecLog is decreased by 39.7%, 78.7%, 80.8%, and 80.2%, respectively. Compared to No-logging, the average latency of DecLog is increased by 5.2%, 8.2%, 9.9%, and 15.2%, respectively. The reasons why DecLog outperforms Beringei in terms of commit latency are multi-fold. Firstly, the log flushing pipeline enables that subsequent insert transactions are not blocked by update transactions. Secondly, the current logging thread only needs to wait for the log entries which have dependencies with the current processed log entries to be persisted by using the thread snapshot. Other logging threads are not blocked at this time, which reduces commit latency. By comparing the experimental results between different workloads, we find that compared with workload A, the average latency of DecLog on workload C and workload D is increased by 8.0% and 12.1%, respectively. The main reason is that with the growing conflicted transactions, the LSN computation overhead incurred through data-driven LSN is increased, which leads to larger log persistence overhead and transaction commit latency.

Recovery. We evaluate the recovery performance of DecLog, DecLog-SSD, Beringei-NVM, and Beringei-SSD on workload A–C. After completing the execution of the three workloads, the database process is terminated. Upon restart, we record the total CPU clock time taken by the recovery process as the recovery time and the results is shown in Table 2. The average recovery time of DecLog is shorter than Beringei-NVM by 80.8%, 80.3%, and 80.6%, respectively. The average recovery time of DecLog-SSD is shorter than Beringei-SSD by 72.6%, 79.3%, and 77.2%, respectively. The reason is that Beringei-SSD and Beringei-NVM can only execute recovery based on centralized LSN in a single thread, while DecLog can achieve multi-threaded recovery by logs, which reduces the recovery time. The recovery for the workload B and workload C is slightly faster than that of workload A. This is because that there are no logs for the transactions that only include read operations. Moreover, when compared with DecLog-SSD, the average recovery time of DecLog

Table 2: Recovery Time

Logging Strategy	Recovery Time of Workloads (s)		
	A	B	C
DecLog	77.7	72.5	72.7
DecLog-SSD	109.0	76.8	85.4
Beringei-NVM	405.6	367.2	375.7
Beringei-SSD	398.3	371.6	374.0
DecLog with checkpoint	3.5	1.9	3.1

is reduced by 28.7%, 5.6%, and 14.9%, respectively. This is mainly due to the inherent performance advantages of NVM. However, the recovery time of Beringei-NVM is comparable with that of Beringei-SSD. It indicates that if we directly apply the centralized logging and recovery design in NVM, it is unable to realize the full potential of NVM performance. Compared with DecLog, the recovery time of DecLog with checkpoint is reduced by 95.5%, 97.4%, and 95.7, respectively. The reason is that only log entries after the offsets recorded in the checkpoint need to be replayed, which alleviates the overhead of recovery.

5.3 Sensitivity Study

We first evaluate the effect of different logging modules in DecLog on database performance by using all workloads. Next, we conduct parameter studies on all workloads to evaluate the impacts of logging thread number, NVM channel, and log queue length, respectively.

Dissecting the Features. To evaluate the impacts of each module in DecLog on database performance, we enable all necessary logging components step-by-step, which include relaxed ordering strategy, data-driven LSN computing protocol, parallel logging, log compression and alignment, and checkpoint. The results by using the strict ordering strategy as the comparative method are presented in Fig. 12(a). We make the following observations: (1) Compared with the strict ordering strategy, the throughput gained by the relaxed ordering strategy is 19.4%, 11.4%, 13.0%, and 12.1%, respectively. The main reason is that relaxed ordering strategy avoids unnecessary *sfence* instructions, which thereby reduces CPU wait latency. (2) After adding the data-driven LSN module on top of relaxed ordering strategy, the throughput increases by 6.7%, 13.2%, 15.9%, and 22.3%, respectively when compared to using only the relaxed ordering strategy. This is because the data-driven LSN computing protocol avoids the contention of LSN by independent transactions. (3) After adding parallel logging, DecLog achieves higher throughput, which is 3.6×, 3.5×, 3.5× and 3.4×, respectively. The main reason is that parallel logging increases the bandwidth for persisting logs in NVM and the tread snapshot alleviates the overhead of synchronization, which improves the logging performance. (4) After adding log compression and alignment module, the transaction throughput decreases by 5.1%, 4.7%, 2.8% and 1.0%, respectively. This is because that log compression consumes additional CPU resources and incurs more computing overhead. However, from the experimental results, we can see that the logging performance loss is relatively small. This is because DecLog aligns log entries by 256 bytes, which in turn improves logging performance. In addition, logging compression reduces the amount of data persisted in NVM,

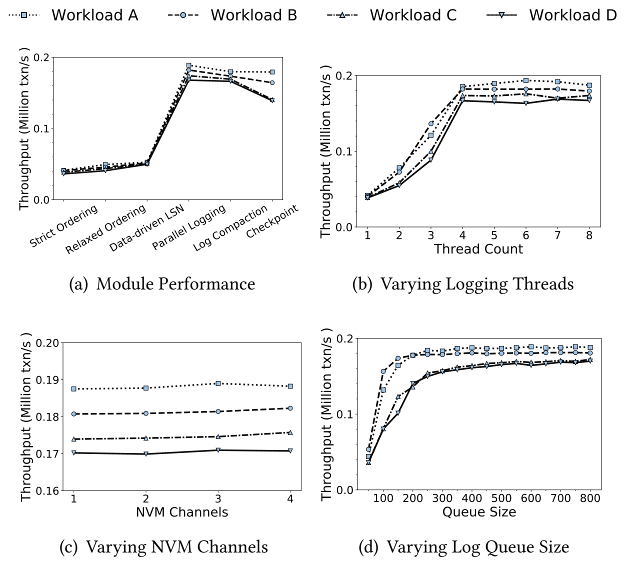


Figure 12: Performance of DecLog

which extends the lifetime of NVM. (5) After adding checkpoint, the transaction throughput is about the same for the workload A. For the workload B–D, the throughput decreases by 5.3%, 16.9%, and 16.4% on average, respectively. This is because the checkpoint consumes CPU and IO resources, which thereby affects the transaction throughput.

Impact of Logging Thread Number. Fig. 12(b) reports the correlation of throughput and the number of logging threads when we fix the transaction thread number to 28 and adjust the logging thread number from 1 to 8. The results show that the throughput increases nearly linearly with the increase of logging thread number, and reaches its peak when the logging thread number is 4. Therefore, DecLog sets the default number of logging threads as 4. By analyzing the results of each workload, we observe that: (1) For the workload A, the throughput gains largely by 4.5× when the logging thread number increases from 1 to 4. This is because the multi-threaded concurrent logging of DecLog improves the logging performance; (2) For the workload C, the throughput is enlarged by 4.5× when the logging thread number increases from 1 to 4. But the throughput is still less than that of workload A. This is because DecLog uses the thread snapshot to synchronize logging threads. When conflicts of transactions exist, thread snapshot does not block the commitment of other transactions with no dependencies, and thus affects the throughput. (3) For the workload D, the throughput improvement is 4.3× when the logging thread number increases from 1 to 4, which is slightly smaller than that of workload C. This is because workload D has more conflicting transactions, and the data-driven LSN computing protocol needs to perform more repeated computations. However, in practical applications, there are very few conflicts among update transactions in TSDBMS, so their impacts on database performance is trivial.

Impact of NVM Channel. In Fig. 12(c), we fix the logging thread number to 4 and vary the NVM channel. We can observe that the throughputs for the four workloads remain steady, as the NVM channel increases from 1 to 4. The results indicate that concurrent writes to different NVM channels do not improve system throughput significantly. This is mainly due to the high random write performance of NVM. On the one hand, the performance of a single NVM channel is already sufficient to handle logs generated by the database transaction module. On the other hand, although increasing the number of NVM channels can improve the logging performance to some extent, DecLog has alleviated the overhead of logging in a single NVM channel. Therefore, the bottleneck of the overall system performance lies in other system modules (e.g., transaction processing and lock contention), which is beyond the scope of this study.

Impact of Log Queue Length. To evaluate the impact of log queue length on database performance, we vary the log queue length from 50 to 800 and fix other parameters as their default values. As shown in Fig. 12(d), with the log queue length increases, the throughput for each workload increases and first reaches its maximum when the length is 300. Then the throughput remains steady. The reasons mainly include two aspects. First, when the log queue length is relatively small, the log entries in each log queue can reside in CPU cache. When the log queue length reaches the threshold, the *clwb* instruction is used to flush log entries from CPU cache to NVM in parallel. With an increasing of the log queue length, the amount of log entries that are flushed by *clwb* instructions to NVM increases. Hence, the throughput of the database is improved. Secondly, when the log queue length exceeds 300, log entries in log queues exceed the CPU cache capacity. When the *clwb* instruction is used to flush data back to NVM, an additional operation is required to copy the data from DRAM to the corresponding NVM address, which increases the persistence overhead. Therefore, the default log queue length of DecLog is set to 300. Moreover, the setting of the log queue length is related to the average size of log entries and the size of CPU cache, so it needs to be adjusted according to the actual situation in use.

6 CONCLUSION AND FUTURE WORK

We design and implement a decentralized logging system DecLog in NVM for TSDBMS by utilizing the characteristics of time series data to improve the logging performance. First, we propose a data-driven LSN based on the data accessed by transactions and track the transaction dependencies. Next, we design a relaxed ordering strategy to persist log entries to NVM by using log flushing pipeline, which effectively addresses the transaction serialization issue caused by a heavy use of *sfence* instructions. Based on these, we propose a thread snapshot based parallel logging method to further improve the concurrency of persisting logs to NVM in multi threads with less thread synchronization overhead. To decrease the log storage footprint and to fully utilize NVM properties, we design a log compression and alignment algorithm to process the logs. In addition, we implement checkpoint and recovery algorithms of DecLog to reduce the recovery time. The experimental study shows that the throughput of DecLog outperforms the Beringei TSDBMS by up to 4.6× with less recovery time in the YSCB-TS benchmark.

In the future, it is of great interest to extend the decentralized logging system in NVM to distributed TSDBMS by exploiting the characteristics of time series data and workloads.

REFERENCES

[1] 2016. NVLog For PostgreSQL. https://www.pgcon.org/2016/schedule/events/945.en.html.

[2] 2017. YCSB-TS. http://tsdbbench.github.io/YCSB-TS/.

[3] 2021. OpenTSDB. http://opentsdb.net/.

[4] 2022. Intel 64 and IA-32 Architectures. https://www.intel.com/content/www/us/en/content-details/671200/intel-64-and-ia-32-architectures-software-developer-s-manual-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.

[5] 2023. Apache Druid. https://druid.apache.org/.

[6] 2023. InfluxDB. https://www.influxdata.com/.

[7] 2023. Intel Persistent Memory Development Kit. https://pmem.io/pmdk/.

[8] 2023. Kdb+. https://kx.com/.

[9] 2023. Optane. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/optane-for-data-centers.html.

[10] 2023. Prometheus. https://prometheus.io/.

[11] 2023. TimescaleDB. https://www.timescale.com/.

[12] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak R. Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. 2013. Scuba: Diving into Data at Facebook. *VLDB* 6, 11 (2013), 1057–1067.

[13] Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George Talbot, Nick Taylor, and Adam Tart. 2020. Monarch: Google's Planet-Scale In-Memory Time Series Database. *VLDB* 13, 12 (2020), 3181–3194.

[14] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *VLDB* 11, 5 (2018), 553–565.

[15] Joy Arulraj, Andrew Pavlo, and Subramanya Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *SIGMOD*. 707–722.

[16] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *VLDB* 10, 4 (2016), 337–348.

[17] Karla L. Caballero Barajas and Ram Akella. 2015. Dynamically Modeling Patient's Health State from Electronic Medical Records: A Time Series Approach. In *SIGKDD*. 69–78.

[18] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD*. 275–290.

[19] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *VLDB* 8, 5 (2015), 497–508.

[20] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *VLDB* 8, 7 (2015), 786–797.

[21] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *ASPLOS*. 1077–1091.

[22] Facebook. 2017. Beringei. https://github.com/facebookarchive/beringei.

[23] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. 2011. High performance database logging using storage class memory. In *ICDE*. 1221–1231.

[24] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *ISCA*. 652–665.

[25] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD*. 243–252.

[26] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD*. 877–892.

[27] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent Memory Hash Indexes: An Experimental Evaluation. *VLDB* 14, 5 (2021), 785–798.

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

[29] Kaisong Huang, Yuliang He, and Tianzheng Wang. 2022. The Past, Present and Future of Indexing on Persistent Memory. *VLDB* 15, 12 (2022), 3774–3777.

[30] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *FAST*. 187–200.

[31] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. *VLDB* 3, 1 (2010), 681–692.

[32] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2012. Scalability of write-ahead logging on multicore and multisocket hardware. *VLDBJ* 21, 2 (2012), 239–263.

[33] Hyungsoo Jung, Hyuck Han, and Sooyong Kang. 2017. Scalable Database Logging for Multicores. *VLDB* 11, 2 (2017), 135–148.

[34] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *FAST*. 191–205.

[35] Jong-Bin Kim, Hyeongwon Jang, Seohui Son, Hyuck Han, Sooyong Kang, and Hyungsoo Jung. 2019. Border-Collie: A Wait-free, Read-optimal Algorithm for Database Logging on Multicore Hardware. In *SIGMOD*. 723–740.

[36] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD*. 1675–1687.

[37] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *ASPLOS*. 385–398.

[38] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *SIGMOD*. 691–706.

[39] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[40] Sangjin Lee, Alberto Lerner, André Ryser, Kibin Park, Chanyoung Jeon, Jinsub Park, Yong Ho Song, and Philippe Cudré-Mauroux. 2022. X-SSD: A Storage System with Native Support for Database Logging and Replication. In *SIGMOD*. 988–1002.

[41] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *SOSP*. 462–477.

[42] Gang Liu, Leying Chen, and Shimin Chen. 2021. Zen: a High-Throughput Log-Free OLTP Engine for Non-Volatile Main Memory. *VLDB* 14, 5 (2021), 835–848.

[43] Haikun Liu, Di Chen, Hai Jin, Xiaofei Liao, Binsheng He, Kan Hu, and Yu Zhang. 2021. A Survey of Non-Volatile Main Memory Technologies: State-of-the-Arts, Practices, and Future Directions. *J. Comput. Sci. Technol.* 36, 1 (2021), 4–32.

[44] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *VLDB* 13, 7 (2020), 1078–1090.

[45] Jian Liu, Kefei Wang, and Feng Chen. 2021. TSCache: An Efficient Flash-based Caching Scheme for Time-series Data Workloads. *VLDB* 14, 13 (2021), 3253–3266.

[46] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *VLDB* 13, 8 (2020), 1147–1161.

[47] Yongping Luo, Peiquan Jin, Qinglin Zhang, and Bin Cheng. 2021. TLBtree: A Read/Write-Optimized Tree Index for Non-Volatile Memory. In *ICDE*. 1889–1894.

[48] Minghua Ma, Shenglin Zhang, Junjie Chen, Jim Xu, Haozhe Li, Yongliang Lin, Xiaohui Nie, Bo Zhou, Yong Wang, and Dan Pei. 2021. Jump-Starting Multivariate Time Series Anomaly Detection for Online Service Systems. In *USENIX ATC*. 413–426.

[49] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys*. 183–196.

[50] C. Mohan. 1999. Repeating History Beyond ARIES. In *VLDB*. 1–17.

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

[52] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *FAST*. 31–44.

[53] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *OSDI*. 1047–1064.

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

[55] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. In *DaMoN*. 8:1–8:7.

[56] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*. 371–386.

[57] Jong-Hyeok Park, Gihwan Oh, and Sang-Won Lee. 2017. SQL Statement Logging for Making SQLite Truly Lite. *VLDB* 11, 4 (2017), 513–525.

[58] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *VLDB* 8, 12 (2015), 1816–1827.

[59] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage Management in the NVRAM Era. *VLDB* 7, 2 (2013), 121–132.

[60] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *USENIX Annual Technical Conference*. 33–48.

[61] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. 18–32.

[62] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *SIGMOD*. ACM, 1541–1555.

[63] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent Memory I/O Primitives. In *DaMoN*. 12:1–12:7.

[64] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *FAST*. 61–75.

[65] Stratis D. Viglas. 2015. Data Management in Non-Volatile Memory. In *SIGMOD*. 1707–1711.

[66] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin Mcgrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. 2020. Apache IoTDB: Time-series database for Internet of Things. *VLDB* 13, 12 (2020), 2901–2904.

[67] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *VLDB* 7, 10 (2014), 865–876.

[68] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2017. Query Fresh: Log Shipping on Steroids. *VLDB* 11, 4 (2017), 406–419.

[69] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. 2014. Using restricted transactional memory to build a scalable in-memory database. In *EuroSys*. 26:1–26:15.

[70] Sung-Ming Wu and Li-Pin Chang. 2022. Rethinking key-value store for byte-addressable optane persistent memory. In *DAC*. 805–810.

[71] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *USENIX Annual Technical Conference*. 349–362.

[72] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. 2020. Taurus: Lightweight Parallel Logging for In-Memory Database Management Systems. *VLDB* 14, 2 (2020), 189–201.

[73] Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jianmin Wang. 2022. Time Series Data Encoding for Efficient Storage: A Comparative Analysis in Apache IoTDB. *VLDB* 15, 10 (2022), 2148–2160.

[74] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Kenry Huang, Xinjun Yang, Wei Cao, and Feifei Li. 2021. Revisiting the Design of LSM-tree Based OLTP Storage Engine with Persistent Memory. *VLDB* 14, 10 (2021), 1872–1885.

[75] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *FAST*. 169–182.

[76] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *FAST*. 167–181.

[77] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD*. 1629–1642.

[78] Baoquan Zhang and David H. C. Du. 2021. NVLSM: A Persistent Memory Key-Value Store Using Log-Structured Merge Tree with Accumulative Compaction. *ACM Trans. Storage* 17, 3 (2021), 23:1–23:26.

[79] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*. 465–477.

[80] Haoren Zhu, Shih-Yang Liu, Pengfei Zhao, Yingying Chen, and Dik Lun Lee. 2022. Forecasting Asset Dependencies to Reduce Portfolio Risk. In *AAAI*. 4397–4404.

[81] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *SIGMOD*. 685–699.

[82] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *OSDI*. 461–476.

[83] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 128:1–128:26.

APPENDIX

We proceed to discuss the implementation details of DecLog.

A DATA UPDATE AND DELETION

As Beringei does not support data updates and deletions, we implement these functionalities based on Beringei to test the performance of DNLog. We utilizes a data update map with timestamp as key and value of data point as value to store updated data. The tombstone bit in extended timestamp is used to indicate whether the data is updated or deleted. When data in a time series is updated, the tombstone flag of the corresponding data is set to 1, and then the updated data is recorded in the data update map, which avoids the overhead of adding read/write locks to the entire time series.

Algorithm 1: *updateDatapoint()*

```
  Input: Datapoint dp
  Output: True or False
1 Init: time series TS, buckets B;
2 seriesID = hashMap(dp.key);
3 if seriesID is NULL then
4     return false;
5 else
6     b = findBucket(dp.timestamp);
7     curTS = TS[seriesID];
8     dpID = findDataPoint(curTS, b, dp.timestamp);
9     if dpID is NULL then
10        return false;
11    else
12        Set the tombstone in curTS[dpID];
13        Record the log entry;
14        Lock map;
15        updateID = findDataPoint(map, dp.key);
16        if updateID then
17            map.update(updateID, dp);
18        else
19            map.insert(dp);
20        Unlock map and unset the lock bit in curTS[dpID];
21        return true;
```

The pseudocode for data updates is presented in Alg. 1, with an input of data point *dp* and an output of execution success status. Initialization is performed for time series and buckets at TSDBMS startup (line 1). For a given data point *dp*, the corresponding time series ID is obtained through hash mapping (line 2). If *seriesID* is null, the updated data does not exist, and the execution failure status is directly returned. If *seriesID* is not null (lines 5–22), the corresponding bucket *b* and time series *curTS* are obtained using the timestamp and *seriesID* of the data point (lines 6–7). The original data point address *dpID* is obtained through binary search in the corresponding bucket and *curTS* (line 8). If *dpID* is null, the updated data does not exist, then the execution failure status is returned directly (lines 9-10). If *dpID* is not null (lines 11–21), first, the lock position of the current data is obtained by using the method described in Sec. 4.1.2, and the tombstone flag is modified (line 12), which indicates that the original data in the current time series is invalid. Then, the corresponding log entry is recorded (line 13). After the log is persisted, the updated data is recorded in the data update map *map* (lines 14–21). Specifically, the lock of *map* is obtained (line 14), and the update record *updateID* of the current data point is searched by using binary search within *map* (line 15). If *updateID* is not null (lines 16–17), it indicates that the update record of *dp* already exists in data update map *map*, and the original data record is updated directly (line 17). Otherwise, a new data update record is inserted in *map* (lines 18–19). Finally, the lock of *map* is released, the lock in time series is reset (line 20), and the execution success status is returned (line 21).

B LOG FLUSHING PIPELINE

We utilize two asynchronous logging threads to implement the log flushing pipeline, and each logging thread processes a log queue. At any time, only one log queue is available for receiving logs, while the other queue is blocked. Transition between the active and blocked queue occurs when a non-zero LSN is encountered. Synchronization between threads is implemented through the use of condition variables, which ensures log entries are flushed to NVM in the LSN order.

The pseudocode for processing logs in a logging thread is presented as Alg. 2. The input for Alg. 2 is a transaction X. First, the log queue is initialized (line 1), and an NVM map is determined (lines 2–3). Subsequently, the LSN of the current log entry $\mathbb{L}$ is computed by using data-driven LSN discussed in Sec. 4.1 (line 4). Next, the log entry are processed by the compression and alignment algorithm as described in 4.4 (line 5). If the combined size of the current log queue and the log entry $\mathbb{L}$ exceeds a predefined threshold *KQueueSize*, or if the LSN of the entry is nonzero (line 6), the $flushQueue()$ function is invoked to persist the current log queue to NVM (lines 7–10). Otherwise, $\mathbb{L}$ is added to the log queue directly (line 12).

Algorithm 2: *logging()*

```
  Input: transaction $X$
  Output: void
1 Init: log queue $q$, log queue size KQueueSize;
2 if !file.mapped() then
3     pmemMapFile();
4 $\mathbb{L}.LSN$ = calculateLSN($X$);
5 Compress the log entry $\mathbb{L}$;
6 if q.length() + $|\mathbb{L}| >$ KQueueSize or $\mathbb{L}.LSN \neq 0$ then
7     if $\mathbb{L}.LSN \neq 0$ then
8         flushQueue($\mathbb{L}$);
9     else
10        flushQueue();
11 if $\mathbb{L}.LSN = 0$ then
12    q.enQueue($\mathbb{L}$);
```

The pseudocode to persist log entries in a log queue to NVM is presented as Alg. 3, with an input of a log entry $\mathbb{L}$. First, NVM address *addr*, log entry alignment size *alignedSize*, and the minimum alignment size *kMinAlignedSize* are initialized (line 1). For each log entry in log queue (lines 2-9), if the log size is close to 256 bytes (line 3), log entries with length *alignedSize* is persisted to NVM (lines 4–5), followed by updating the NVM address (line 6). Otherwise, the alignment length *alignedSize* is updated (lines 7-9). The $sfence()$ function is used to avoid CPU instruction reordering (line 10). If the input log entry is not empty (lines 11–20), indicating an update transaction log entry, the thread waits for log entries in the other log queue with a conditional variable (line 12), followed by calling $treadSync()$ to perform multi-threaded logging synchronization according to the method described in Sec. D. Next, the log entry is persisted to NVM and the $sfence()$ function is used to ensure log sequence (lines 14-18). Finally, the thread snapshot is updated (line 19) and the other log queue is notified that the update transaction log entry has been successfully persisted in NVM by using a conditional variable (line 20).

Algorithm 3: *flushQueue()*

```
Input: log entry $\mathbb{L}$
Output: void
1  Init: NVM address addr, alignedSize = 0, minimum
     alignment size kMinAlignedSize;
2  foreach elem in queue do
3      if $alignedSize\%256 > kMinAlignedSize$ then
4          memcpy(addr, elems, alignedSize);
5          clwb(addr, alignedSize);
6          addr += alignedSize;
7      else
8          elems.append(elem);
9          alignedSize += elem.size();
10 sfence();
11 if log != NULL then
12     conditionVarible.wait();
13     threadSync($\mathbb{L}$.LSN);
14     memcpy(addr, $\mathbb{L}$, $|\mathbb{L}|$);
15     clwb(addr, $|\mathbb{L}|$);
16     sfence();
17     addr += $|\mathbb{L}|$;
18     sfence();
19     update snapshot;
20     conditionVarible.notify();
```

C DATA-DRIVEN LSN

The implementation of data-driven LSN consists of LSN computation and extended timestamp update. LSN is computed based on the expended timestamp of the data. If it is an insert transaction, the LSN of the transaction log entry is set to 0. Otherwise, the maximum LSN is assigned as the LSN of the current transaction log, and *uts* of all updated data within the transaction is updated to the LSN.

The pseudocode for computing LSN is shown as Alg. 4, with an input of transaction X and an output of *LSN*. First, the maximum LSN *maxLSN* is initialized to 0 (line 1). For each operation in transaction X (line 2), if the current operation is a data insert (line 4), the LSN is directly returned as 0 (lines 5–6). Otherwise, *uts* in extended timestamp is read (line 7). Next, the LSN is computed by *uts* (line 8). If LSN reaches a given threshold (lines 9–11), the checkpoint procedure is called (line 10), then the LSN of this transaction log is set as 1 (line 11). Otherwise, the maximum timestamp *maxLSN* is updated (line 12). Finally, the LSN of the current transaction log is returned (line 13).

Algorithm 4: *computeLSN()*

```
Input: transaction $X$
Output: LSN
1  Init: maxLSN = 0;
2  foreach o in X do
3      LSN = 0;
4      if o.type = insertion then
5          LSN = 0;
6          continue;
7      $uts = (o.timestamp << 52) \& 0x3FF + 1$;
8      $LSN = uts + 1$;
9      if $LSN = 1024$ then
10         checkpoint();
11         return 1;
12     $maxLSN = maxLSN > LSN \; ? \; maxLSN : LSN$;
13 return maxLSN;
```

D PARALLEL LOGGING

The parallel logging module mainly focuses on concurrent logging and synchronization among multiple logging threads. This is crucial to ensure that the order of log flushing is consistent with the order of transaction commits.

The pseudocode for parallel logging is presented as Alg. 5, with an input of transaction X. First, the logging threads and local variable *id* are initialized (line 1). When the transaction is ready to commit (line 2), the keys of time series data accessed by the transaction are hashed (line 3), and the logging thread responsible for processing the current transaction is determined based on the initialized logging thread number (line 4), and the $logData()$ function is called to record the log entry (line 5). At last, the current transaction is committed (line 6).

Algorithm 5: *Parallel Logging*

```
Input: transaction $X$
Output: void
1  Init: logging threads, id = 0;
2  prepareCommit(X); // OCC protocol
3  id = hashMap(X);
4  logThread = threads[id % KThreads];
5  logThread.logData(X);
6  commit(X);
```

The pseudocode for log synchronization is shown as Alg. 6. The LSN of the log entry that is processed by the current logging thread is the input for the algorithm. Firstly, the update transaction log quantity *curThreadNums* recorded by other threads in the thread snapshot is initialized, and the state flag is set to true (line 1). The log synchronization process requires to read the thread snapshot in a loop manner (lines 2-11). For the log being processed in the current logging thread (lines 3-11), if *threadLSN* is greater than or equal to the current LSN (line 4), it indicates that there is no dependency between the the current logging thread and the log entry being processed. Therefore, the *state* is set to false (line 5). Otherwise, it is necessary to further determine the update transaction log quantity being processed by the current logging thread (lines 6-11). If the initialized update transaction log quantity *curThreadNum* is less than the number of logs currently recorded in the thread snapshot *threadNum* (line 7), it indicates that the current logging thread has completed the previous update transaction logs persistence, so the *state* is set to false (line 8). Otherwise, it indicates that the current log has not been persisted to NVM yet, so the *state* is set to true (line 10), and it is processed in the next iteration (line 11).

Algorithm 6: *threadSync()*

```
   Input: LSN
   Output: void
1  Init: curThreadNums = snapshot.threadNums,state = true;
2  while state do
3      foreach threadLSN, threadNum in snapshot do
4          if threadLSN ≥ LSN then
5              state = false;
6          else
7              if curThreadNum < threadNum then
8                  state = false;
9              else
10                 state = true;
11                 break;
```

E LOG COMPRESSION AND ALIGNMENT

For each log entry, if the active hash table does not contain the key recorded in the entry, the key is added to the hash table. Otherwise, dictionary encoding is applied to compress the key field. Further, delta-of-delta and XOR compression algorithms are used for integer fields and floating-point fields, respectively. Finally, compressed log entry is aligned by 256 bytes and transactions are committed in group.

Alg. 7 presents the pseudocode for log compression and alignment. The minimum alignment size *minSize* is initialized at TS-DBMS startup (line 1). We take the timestamp $\mathbb{L}.timestamp$ and floating-point number $\mathbb{L}.value$ in a log entry $\mathbb{L}$ to illustrate the delta-of-delta and XOR compression algorithms. Let t_{pre} be the previous timestamp, δ_{pre} be the previous delta number, $value_{pre}$ be the previous floating-point number, ld_{pre} be the previous leading zeros of $value_{pre}$, and tl_{pre} be the previous trailing zeros of $value_{pre}$, which are read from the active hash table by the key of $\mathbb{L}$ (line 2). For the timestamp $\mathbb{L}_{timestamp}$, we compute the delta-of-delta value δ by δ_{pre}, t_{pre}, and δ_{pre} (line 3). Then the encoding is divided into two cases (lines 4–7). If δ is not zero, the timestamp is encoded by δ. Otherwise, we encode it by an equal flag, which indicates that the $\mathbb{L}.timestamp$ is equal to the previous timestamp. For the floating-point number $\mathbb{L}.value$, we first compute the number of leading zeros ld (line 8) and the number of trailing zeros tr (line 9). Next, the XOR value v_{xor} is computed by $\mathbb{L}.value$ and $value_{pre}$ (line 10). We divide the encoding of $\mathbb{L}.value$ into two cases. If ld and tl is equal to the corresponding values ld_{pre} and tl_{pre} in hash table, respectively (line 11), we encode $\mathbb{L}.value$ with v_{xor} (line 11–12), which indicates that the numbers of leading zeros and trailing zeros are equal to those in hash table. Otherwise, ld, tr, and v_{xor} should be used for encoding $\mathbb{L}.value$ (lines 13–14). After all the fields of $\mathbb{L}$ have been compressed (line 15), we update the corresponding values in hash table (line 16) and buff the log entry $\mathbb{L}$ in *buff*. Whenever the buffer size approaches *minSize*, we persist the log entries in *buff* to NVM (line 19) and the corresponding transactions can be committed in group (line 20).

Algorithm 7: *Compression and Alignment*

```
   Input: A log entry 𝕃
   Output: void
1  Init: Minimum alignment size minSize;
2  Read $t_{pre}$, $\delta_{pre}$, $value_{pre}$, $ld_{pre}$, and $tl_{pre}$ from the active hash
     table;
3  $\delta = \mathbb{L}.timetamp - t_{pre} - \delta_{pre}$;
4  if $\delta$ then
5      Encode timestamp with $\delta$;
6  else
7      Encode timestamp with an equal flag;
8  Compute number of leading zeros $ld$;
9  Compute number of trailing zeros $tr$;
10 Compute XOR value $v_{xor}$ of $\mathbb{L}.value$ and $value_{pre}$;
11 if $ld = ld_{pre}$ and $tl = tl_{pre}$ then
12     Encode floating-point number $\mathbb{L}.value$ with $v_{xor}$;
13 else
14     Encode floating-point number $\mathbb{L}.value$ with $ld$, $tr$, and
         $v_{xor}$;
15 Compress other fields of 𝕃;
16 Update hash table;
17 Buffer the log entry 𝕃 in buff;
18 if |buff| > minSize then
19     Persist log entries in buff to NVM;
20     Commit corresponding transactions in group.
```

F CHECKPOINT

The checkpoint implements a sequential process. Whenever the LSN of a log entry reaches a given threshold, the checkpoint process is triggered, the update transactions are blocked, and the subsequent insert transactions can continue to execute.

Alg. 8 shows the psuedocode for checkpoint that contains four steps as follows: (1)We block update transactions (line 1), record the start time t_1 of checkpoint (line 2), and write dirty data back to permanent storage. (2) After log entries in each log buffers have been flushed to NVM (line 4), we record the offsets in each log bock (lines 5–6) and purge outdated logs (line 7). (3) All the extended timestamps are set to 0s (line 8) and the completion time t_{LSN} is recorded in the checkpoint log (line 9). (4) We persist a copy of the active hash table in NVM (line 10) and reset fields except for the string encoding (line 11). After the end time t_2 has been persisted in NVM (line 12), we allow update transactions to execute (line 13).

Algorithm 8: *Checkpoint*

```
   Input: Start time $t_1$.
   Output: Checkpoint log.
1  Block update transactions;
2  Record start time $t_1$;
3  Write dirty data to permanent storage;
4  Flush log buffers;
5  foreach block B in log blocks do
6      Record the offset of B;
7  Purge outdated logs;
8  Reset extended timestamps;
9  Record completion time $t_{LSN}$;
10 Copy and persist the active hash table in NVM;
11 Reset active hash table;
12 Record end time $t_2$;
13 Allow update transactions.
```

G RECOVERY

The recovery module mainly consists of log parsing and log redo. For each parsed log, a redo operation is performed for insert transactions. Afterwards, log entries for update operations are replayed in the order of LSN.

The recovery process needs to be handled differently based on checkpoint and non-checkpoint scenarios. Since the recovery for non-checkpoint scenario is similar with the other one, here we present the pseudocode for recovery with checkpoint log in Alg. 9. At TSDBMS startup, we initialize one thread for log parsing, one thread to replay logs of update transactions, and multiple threads to replay logs of insert transactions (line 1). We first read the latest checkpoint log (line 2) and the active hash table map_2 (line 3). Next, all the log blocks are parsed from the offsets recorded in the checkpoint log (line 4–25). The address $addr_j$ of the current processed log block is initialized as $addr_i$ (line 5). Since the log entries are aligned by 256 bytes, we parse entries by 256-byte data block (lines 6–26). The log entries are first buffered in *buff* (line 7). If an end flag of the data block is encountered, $addr_j$ is accumulated by 256 (lines 8–9). Otherwise, the next log entry is read from the data block (lines 10–11). Let l be the length of log queues. When l consecutive 256-byte data blocks are 0s, log parsing for the current log block has completed (lines 12–13). Otherwise, log entries in *buff* are parsed (line 15–26). We first decompress the timestamp $\mathbb{L}.timestamp$. Next, the log entry $\mathbb{L}$ is decompressed by map_1 or map_2 (lines 17–22). If the log entry is for a insert transaction (line 21), we repay it directly (line 22). Otherwise, the log entry is buffered in the *LSN*-th bucket b_1^{LSN} or b_2^{LSN} which is differentiated by $\mathbb{L}.timestamp$ and t_{LSN} (lines 24–27). After all logs have been parsed, log entries in b_1 and b_2 are replayed in the order of LSN, respectively (line 28-29).

Algorithm 9: *Recovery*

```
   Input: logs
   Output: data in logs
1  Init:one thread for log parsing, one thread to replay logs of
     update transactions, and multiple threads to replay logs of
     insert transactions;
2  Read $t_1$, $t_{LSN}$, $t_2$, hash table $map_1$, and offsets from the
     latest checkpoint log;
3  Read the active hash table $map_2$ from NVM;
4  foreach offset $addr_i$ in log block do
5      $addr_j = addr_i$;
6      while 256-byte data block is completed do
7          Read 256-byte aligend log entries to buffer;
8          if end of data block then
9              $addr_j += 256$;
10         else
11             Read the next log entry;
12         if l consecutive 256-byte data blocks are 0s then
13             break;
14         else
15             foreach log entry $\mathbb{L}$ in buffer do
16                 Decompress $\mathbb{L}.timestamp$;
17                 if $\mathbb{L}.timestamp < t_2$ then
18                     Decompress $\mathbb{L}$ by $map_1$;
19                 else
20                     Decompress $\mathbb{L}$ by $map_2$;
21                 if $\mathbb{L}.LSN = 0$ then
22                     Replay the log entry;
23                 else
24                     if $\mathbb{L}.timestamp < t_{LSN}$ then
25                         Buffer the entry in bucket $b_1^{LSN}$;
26                     else
27                         Buffer the entry in bucket $b_2^{LSN}$
28 Replay log entries in $b_1$ in the order of LSN;
29 Replay log entries in $b_2$ in the order of LSN.
```