

## MIT Open Access Articles

### *Taurus: lightweight parallel logging for in-memory database management systems*

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

**Citation:** Xia, Yu, Yu, Xiangyao, Pavlo, Andrew and Devadas, Srinivas. 2020. "Taurus: lightweight parallel logging for in-memory database management systems." Proceedings of the VLDB Endowment, 14 (2).

**As Published:** 10.14778/3425879.3425889

**Publisher:** VLDB Endowment

**Persistent URL:** <https://hdl.handle.net/1721.1/143468>

**Version:** Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

**Terms of use:** Creative Commons Attribution-Noncommercial-Share Alike



# Taurus: Lightweight Parallel Logging for In-Memory Database Management Systems (Extended Version)

Yu Xia

Massachusetts Institute of Technology  
yuxia@mit.edu

Andrew Pavlo

Carnegie Mellon University  
pavlo@cs.cmu.edu

Xiangyao Yu

University of Wisconsin–Madison  
yxy@cs.wisc.edu

Srinivas Devadas

Massachusetts Institute of Technology  
devadas@mit.edu

## Abstract

Existing single-stream logging schemes are unsuitable for in-memory database management systems (DBMSs) as the single log is often a performance bottleneck. To overcome this problem, we present Taurus, an efficient parallel logging scheme that uses multiple log streams, and is compatible with both data and command logging. Taurus tracks and encodes transaction dependencies using a vector of log sequence numbers (LSNs). These vectors ensure that the dependencies are fully captured in logging and correctly enforced in recovery. Our experimental evaluation with an in-memory DBMS shows that Taurus’s parallel logging achieves up to 9.9× and 2.9× speedups over single-streamed data logging and command logging, respectively. It also enables the DBMS to recover up to 22.9× and 75.6× faster than these baselines for data and command logging, respectively. We also compare Taurus with two state-of-the-art parallel logging schemes and show that the DBMS achieves up to 2.8× better performance on NVMe drives and 9.2× on HDDs.

## 1 Introduction

A database management system (DBMS) guarantees that a transaction’s modifications to the database persist even if the system crashes. The most common method to enforce durability is *write-ahead-logging*, where each transaction sequentially writes its changes to a persistent storage device (e.g., HDD, SSD, NVM) before it commits [29]. With increasing parallelism in modern multicore hardware and the rising trend of high-throughput in-memory DBMSs, the scalability bottleneck caused by sequential logging [16, 35, 37, 43] is onerous, motivating the need for a parallel solution.

It is non-trivial, however, to perform parallel logging because the system must ensure the correct recovery order of transactions. Although this is straightforward in sequential logging because the LSNs (the positions of transaction records in the log file) explicitly define the order of transactions, it is not easy to efficiently recover transactions that are distributed across multiple logs without central LSNs. A parallel logging scheme must maintain transactions’ order information across multiple logs to recover correctly.

There are several parallel logging and recovery proposals in the literature [16, 35, 37, 43]. These previous designs, however, are limited in their scope and applicability. Some algorithms support only parallel data logging but not parallel command logging [14, 35, 43]; some can only parallelize the recovery process but not the logging process [8, 30]; a few protocols assume NVM hardware but do not work for conventional storage devices [3, 4, 6, 10, 15, 21,

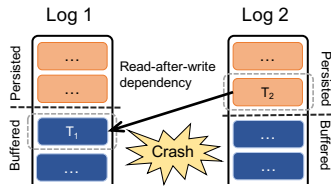
22, 36]. As such, previously proposed methods are insufficient for modern DBMSs in diverse operating environments.

To overcome these limitations, we present **Taurus**, a lightweight protocol that performs both logging and recovery in parallel, supports both data and command logging, and is compatible with multiple concurrency control schemes. Taurus achieves this by tracking the inter-transaction dependencies. The recovery algorithm uses this information to determine the order of transactions. Taurus encodes dependencies into a vector of LSNs, which we define as the *LSN Vector (LV)*. LSN Vectors are inspired by vector clocks to enforce partial orderings in message-passing systems [11, 27]. To reduce the overhead of maintaining LVs, Taurus compresses the vector based on the observation that a DBMS can recover transactions with no dependencies in any order. Thus, Taurus does not need to store many LVs, thereby reducing the space overhead.

We compare the performance of Taurus to a sequential logging scheme (with and without RAID-0 setups) and state-of-the-art parallel logging schemes (i.e., Silo-R [35, 43] and Plover [44]) on YCSB and TPC-C benchmarks. Our evaluation on eight NVMe SSDs shows that Taurus with data logging outperforms sequential data logging by 9.9× at runtime, and Taurus with command logging outperforms the sequential command logging by 2.9×. During recovery, Taurus with data logging and command logging are 22.9× and 75.6× faster than the serial baselines, respectively. Taurus with data logging matches the performance of the other parallel schemes, and Taurus with command logging is 2.8× faster at both runtime and recovery. Another evaluation on eight HDDs shows that Taurus with command logging achieves 9.2× and 6.4× faster than these parallel algorithms in logging and recovery, respectively.

The main contributions of this paper include:

- We propose the Taurus parallel scheme that supports both command logging and data logging. We formally prove the correctness and liveness in Appendix A.
- We propose optimizations to reduce the memory footprint of the dependency information that Taurus maintains and extensions for supporting multiple concurrency control algorithms.
- We evaluate Taurus against sequential and the parallel logging schemes, and demonstrate its advantages and generality.
- We open source Taurus and evaluation scripts at [https://github.com/yuxiamit/DBx1000\\_logging](https://github.com/yuxiamit/DBx1000_logging).



**Figure 1: Data Dependency in Parallel Logging** — Transaction  $T_2$  depends on  $T_1$ . The two transactions write to different logs.

## 2 Background

We first provide an overview of conventional serial logging protocols and then discuss the challenges of extending a logging algorithm to support a parallel environment.

### 2.1 Serial Logging

In a serial logging protocol, the DBMS constructs a single log stream for all transactions. The protocol maintains the ordering invariant that, if  $T_2$  depends on  $T_1$ , then the DBMS writes  $T_2$  to disk after  $T_1$ . The DBMS ensures a transaction commits only after it successfully writes the transaction’s log records to disk. During recovery, the DBMS reads the log sequentially, starting from the last checkpoint. The DBMS replays each transaction sequentially until it encounters an incomplete log record or the end of the file.

In general, there are two categories of logging schemes. The first is *data logging*, where log records contain the physical modifications that transactions made to the database. The recovery process with this scheme is to re-apply these changes back to the database. The other category, called *command logging* [26], reduces the amount of log data by only recording transactions’ high-level commands (i.e., invocations of stored procedures). The log records for these commands are typically smaller in size than the physical changes made to the database. The recovery process involves more computation, as all transactions are re-executed sequentially. When the disk bandwidth is the bottleneck, command logging can substantially outperform data logging.

Although serial logging is inherently sequential, one can improve its performance by using RAID disks that act as a single storage device to increase disk bandwidth [31]. Serial logging can also support parallel recovery if the DBMS uses data logging [33, 35, 43]. But the fundamental property that distinguishes serial logging from parallel logging is that it relies on a single log stream that respects all the data dependencies among transactions. On a modern in-memory DBMS with many CPU cores, such a single log stream is a contention point that becomes a scalability bottleneck [35]. Competing for the single atomic LSN counter inhibits performance due to cache coherence traffic [42].

### 2.2 Parallel Logging Challenges

Parallel logging allows transactions to write to multiple log streams (e.g., one stream per disk), thereby avoiding serial logging’s scalability bottlenecks to satisfy the high throughput demands of in-memory DBMSs. Multiple streams inhibit an inherent natural ordering of transactions. Therefore, other mechanisms are required to track and enforce the ordering among these transactions. Fig. 1 shows an example with transactions  $T_1$  and  $T_2$ , where  $T_2$  depends on  $T_1$  with a read-after-write (RAW) data dependency.

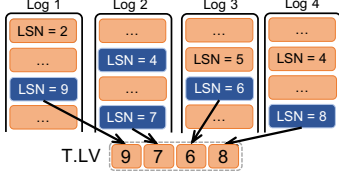
In this example, we assume that  $T_1$  writes to *Log 1* and  $T_2$  writes to *Log 2* and they may be flushed in any order. If  $T_2$  is already persistent in *Log 2* while  $T_1$  is still in the log buffer (shown in Fig. 1), the DBMS must *not* commit  $T_2$  since  $T_1$  has not committed. Furthermore, if the DBMS crashes, then the recovery process must be aware of such data dependency and therefore should not recover  $T_2$ . Specifically, parallel logging faces the following three challenges.

**Challenge #1 – When to Commit a Transaction:** The DBMS can only commit a transaction if it is persistent and all the transactions that it depends on can commit. In serial logging, this requirement is satisfied if the transaction itself is persistent, indicating all the preceding transactions are also persistent. In parallel logging, however, a transaction must identify when other transactions that it depends on can commit, especially those that are storing their log records on other log streams. For the example shown in Fig. 1, the DBMS can commit  $T_2$  only after  $T_1$  is already persistent.

**Challenge #2 – Whether to Recover a Transaction:** A technique like *Early-Lock-Release* (ELR) prevents transactions from waiting for log persistency during execution by allowing a transaction to release locks early before the log records hit disks [8]. But this means that during recovery, the DBMS has to determine whether transactions successfully committed before a crash. The DBMS ignores any transaction that fails to complete properly. For the example in Fig. 1, if  $T_2$  is in the log but  $T_1$  is not, then the DBMS should not process  $T_2$  during recovery.

**Challenge #3 – Determine the Recovery Order:** The DBMS must recover transactions in the order that respects data dependencies. If both  $T_1$  and  $T_2$  are persistent and have committed before the crash, the DBMS must recover  $T_1$  before  $T_2$ , since  $T_2$  reads the value that is written by  $T_1$ .

One can resolve some of the above issues if the DBMS satisfies certain assumptions. For example, if the concurrency control algorithm enforces dependent transactions to write to disks in the corresponding order, then this solves the first and second challenges: the persistence of one transaction implies that any transactions that it depends on are also persistent. If the DBMS uses data logging, then it needs to handle write-after-write (WAW) dependencies, but not read-after-write (RAW) or write-after-read (WAR) dependencies. For example, consider a transaction  $T_1$  that writes  $A=1$ , and a transaction  $T_2$  that reads  $A$  and then writes  $B=A+1$ . Suppose the initial value of  $A$  is 0, and the DBMS schedules  $T_2$  before  $T_1$ , resulting in  $A=1$  and  $B=1$ . With this schedule,  $T_1$  has a WAR dependency on  $T_2$ . If the DBMS does not track WAR dependencies and perform command logging, running  $T_1$  before  $T_2$  will result in  $A=1$  and  $B=2$ , which violates correctness. But if the DBMS performs data logging, then  $T_1$  will have a record of  $A=1$  and  $T_2$  will have a record of  $B=1$ . Regardless of the recovery order between  $T_1$  and  $T_2$ , the resulting state is always correct. Supporting only data logging simplifies the protocol [35, 43]. These assumptions, however, would hurt either performance or generality of the DBMS. Our experiments in Sec. 5 show that Taurus command logging outperforms all the data logging baselines by up to 6.4× in both logging and recovery.



**Figure 2: LSN Vector (LV) example** – The  $i^{\text{th}}$  element of transaction  $T$ 's LV is an LSN of log $_i$ , indicating that  $T$  depends on one or more transactions (rendered in dark blue) in log $_i$  before that LSN.

### 3 Taurus Parallel Logging

We now present the Taurus protocol in detail. The core idea of Taurus is to use a lightweight dependency tracking mechanism called *LSN Vector*. After first describing LSN Vectors, we then explain how Taurus uses them in Sec. 3.2 and Sec. 3.3 during runtime and recovery operations, respectively. We then discuss how Taurus supports index operations like range scan, insertions, and deletions. Lastly, we describe limitations of Taurus and potential solutions.

Although Taurus supports multiple concurrency control schemes (see Sec. 4.3), for the sake of simplicity, we assume strict two-phase locking (S2PL) in this section unless otherwise stated. We also assume that the DBMS uses multiple disks with each log file residing on one disk. Each transaction writes only a single log entry to one log file at commit time. The design of a single log entry per transaction simplifies the protocol and is used by other in-memory DBMSs, including Hekaton [9], Silo [35, 43], and H-Store [19].

#### 3.1 LSN Vector

An *LSN Vector* (LV) is a vector of LSNs that encodes the dependencies between transactions. The DBMS assigns it to either (1) a transaction to record its dependency information or (2) a data item to capture the dependencies between transactions accessing it. The dimension of an LV is the same as the number of logs. Each element of LV indicates that a transaction  $T$  may depend on transactions before a certain position in the corresponding log. Specifically, given a transaction  $T$  and its assigned LV:  $T.LV = (LV[1], LV[2], \dots, LV[n])$ , for any  $1 \leq i \leq n$ , the following property holds:

**PROPERTY 1.** *Transaction  $T$  does not depend on any transaction  $T'$  that maps to the  $i$ -th log with  $LSN > LV[i]$ .*

Fig. 2 shows the LV of an example transaction  $T$ . The second element in  $T.LV$  is 7, meaning that  $T$  may depend on any transaction that maps to *Log 2* with an  $LSN \leq 7$  but no transaction with an  $LSN > 7$ . In this example,  $T$  depends on two transactions in *Log 2* with both LSNs no greater than 7. The semantics of LV is similar to vector clocks [11, 27]. In particular, the following two operations will be frequently used on LVs: *ElemWiseMax* and *comparison*. The *ElemWiseMax* is the element-wise maximum function:

$$LV = \text{ElemWiseMax}(LV', LV'') \Rightarrow \forall i, LV[i] = \max(LV'[i], LV''[i])$$

For *comparison*, the relationships are defined as follows:

$$LV \leq LV' \iff \forall i, LV[i] \leq LV'[i].$$

Following the semantics of vector clocks, LV captures an approximation of the partial order among transactions – LVs of dependent transactions are always ordered and LVs of independent transactions may or may not be ordered.

An LV of a transaction is written to the log together with the rest of the log entry. The dependency information captured by the

partial order is sufficient for resolving the three challenges from Sec. 2.2. Taurus's LVs address these challenges in the following way: (1) A transaction  $T$  can commit if it is persistent and each log has flushed to the point specified by  $T.LV$ , indicating that all transactions that  $T$  depends on are persistent. (2) During recovery, the DBMS determines that a transaction  $T$  has committed before the crash if each log has flushed to the point specified by  $T.LV$ . (3) The recovery order follows the partial order specified by LVs, and the DBMS can recover unordered transactions in parallel.

#### 3.2 Logging Operations

The Taurus protocol is based on *workers* and *log managers* (denoted as  $L_1, L_2, \dots, L_n$ ). Each log manager writes to a unique log file. Each worker is assigned to a log manager and we assume every log manager has exactly  $p$  workers. The log managers and workers run on separate threads. We first describe the protocol's internal data structures and then explain its algorithms.

**Data Structures:** On top of a conventional 2PL protocol, Taurus adds the following data structures to the system.

- *T.LV* – Each transaction  $T$  contains a *T.LV* encoding its dependency as discussed in Sec. 3.1. When  $T$  initially starts, *T.LV* is a vector of zeroes.
- *Tuple.readLV/writeLV* – Each tuple contains two LVs that serve as a medium for transaction LVs to propagate between transactions. Intuitively, these vectors are the maximum LV of transactions that have read/written the tuple. Initially, all elements are zeroes. This does not necessarily incur extra linear storage because the DBMS maintains this metadata in its lock table (cf. Sec. 4.1).
- *L.logLSN* – The highest position in the log file that has not been allocated for log manager  $L$ . It is initialized as zero. Workers reserve space for log records by incrementing *L.logLSN*.
- *L.allocatedLSN* – A vector of length  $p$  that stores the last LSN allocated by each worker of log manager  $L$ . Initially, all elements of *allocatedLSN* are  $\infty$ .
- *L.filledLSN* – A vector of length  $p$ , storing the last LSN filled by each worker of log manager  $L$ . Initially, all elements are zeroes. The purpose of *L.allocatedLSN* and *L.filledLSN* is to determine the point to which the log manager  $L$  can safely flush its log. They are irrelevant to the idea of LV but are important for the correctness of Taurus.
- *Global.PLV* – *PLV* stands for *Persistent LSN Vector* that is a global vector of length  $n$ . The element *PLV<sub>i</sub>* denotes the LSN that log manager  $L_i$  has successfully flushed up to.

**Worker Threads:** Worker threads track dependencies by enforcing partial orders on the LSN Vectors. The logic of a worker thread is contained in the *Lock* and *Commit* functions shown in Alg. 1. The 2PL locking logic is in the *FetchLock* function (Line 2); Taurus supports any variant of 2PL (e.g., deadlock-detection, no-wait, wait-and-die). After a worker thread acquires a lock, it executes Lines 3–5 to update the LV of the transaction. It first updates *T.LV* to be the element-wise maximum of the current *T.LV* and the tuple's *writeLV* (Line 3). This enforces *T.LV* to be no less than the LV of previous writing transactions. If the access is a write, it also updates *T.LV* using the tuple's *readLV*.

The DBMS calls the *Commit* function shown in Lines 6–18 when the transaction finishes. At this moment,  $T$  has locked tuples it accessed. Since the DBMS updates  $T.LV$  for each access, it already captures  $T$ 's dependency information. The DBMS first checks if  $T$  is read-only, and skip generating log records if so. Otherwise, it creates the log record for transaction  $T$  (Line 8). The log record contains two parts: the *redo log* and a copy of  $T$ 's  $LV$  at this moment. The contents of the redo log depends on the logging scheme: the keys and values that  $T$  modified (for data logging), or the information sufficient to reconstruct  $T$  (for command logging). The DBMS writes the record into the corresponding log manager's buffer by *WriteLogBuffer* (Line 10). The algorithm then updates the  $i$ -th dimension of  $T.LV$  to the returned LSN (Line 11), thereby allowing future transactions to capture their dependencies on  $T$ . This update only changes  $T.LV$ , while the copy of  $T.LV$  in the buffer does not contain this update. Lines 12–17 update the *readLV* and/or *writeLV* of each tuple that  $T$  accessed before releasing the locks on those tuples. If  $T$  reads a tuple, it updates the tuple's *readLV* using  $T.LV$ , indicating that the tuple was read by  $T$  and future transactions must respect this dependency. Similarly, if  $T$  has written a tuple, the tuple's *writeLV* is updated accordingly. Updating the LVs and releasing the lock must be executed in an atomic section, otherwise multiple transactions simultaneously updating the *readLV* can cause race conditions leading to incorrect dependencies. As most 2PL protocols use latches to protect the release function, updating LVs can be piggybacked within those latches. For simplicity, we present a long atomic section covering Lines 12–17 (shaded in gray).

After the DBMS releases transaction  $T$ 's locks, it has to wait for  $PLV$  to catch up such that  $PLV \geq T.LV$  (indicating  $T$  is durable). All transactions within the same log manager commit sequentially. Since each log manager flushes records sequentially, this does not introduce a scalability bottleneck. We employ the ELR optimization [8] to reduce lock contention by allowing transactions to release locks before they are durable.

The *Commit* function calls *WriteLogBuffer* (Lines 19–24) to write a log entry into the log buffer. It first allocates space in the log manager's ( $L_i$ ) buffer by atomically incrementing its LSN by the size of the log record (Line 21). It then copies the log record into the log buffer (Line 22). Lines 20 and 23 are indicators for the log manager to decide up to which point it can flush the log buffer to disk. Specifically, before a transaction increments the LSN, it notifies the log manager ( $L_i$ ) that its allocated space is no earlier than its current LSN (Line 20). This leads to  $allocatedLSN[j] \geq filledLSN[j]$ , which instructs  $L_i$  that the log buffer contents after  $allocatedLSN[j]$  are unstable and should not be flushed to the disk. After the log buffer is filled, the transaction updates  $L_i.filledLSN[j]$  so that  $allocatedLSN[j] < filledLSN[j]$ , indicating that the worker thread has no ongoing operations on the log buffer.

To demonstrate how Taurus tracks dependencies, we use the example in Fig. 3 with three transactions ( $T_1, T_2, T_3$ ) and two database objects A,B. WLOG, we assume  $T_1$  and  $T_2$  are assigned to Log 1 and  $T_3$  is assigned to Log 2. In the beginning, A has a *writeLV* [4,2] and a *readLV* [3,7] while object B has [8,6] and [5,11]. ① The DBMS initializes the transactions' LVs as [0,0]. ②  $T_1$  acquires an exclusive lock on A and writes to it. Then,  $T_1$  updates  $T_1.LV$  to be the element-wise maximum among A.*writeLV*, A.*readLV*, and  $T_1.LV$ . In this example,  $T_1.LV = [\max(4,3,0), \max(2,7,0)] = [4,7]$ . Enforcing

**Algorithm 1: Worker Thread** – We assume the worker is the  $j$ -th worker for log manager  $L_i$ .

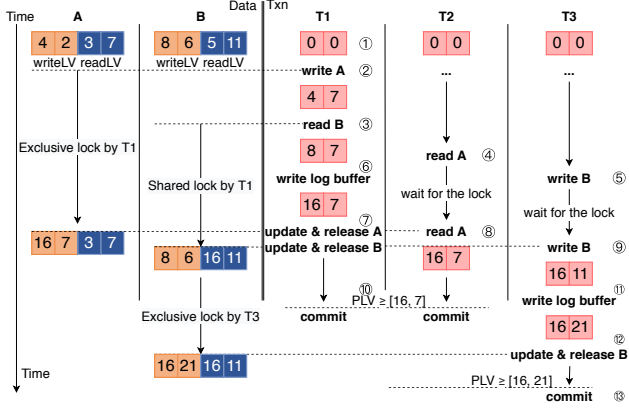
```

1 Function Lock(key, type, T)
   # Lock the tuple following the 2PL protocol.
2   FetchLock(key, type, T);
3   T.LV = ElemWiseMax(T.LV, DB[key].writeLV);
4   if type is write then
5     T.LV = ElemWiseMax(T.LV, DB[key].readLV);
6 Function Commit(T)
7   if T is not read-only then
8     # Include T's LV into the log record.
9     logRecord = {CreateLogRecord(T, copy(T.LV));
10    recordSize = GetSize(logRecord);
11    LSN = WriteLogBuffer(logRecord, recordSize);
12    T.LV[i] = LSN# Update T.LV[i] in the memory;
13   for key  $\in$  T's access set do
14     if T reads DB[key] then # Atomic Section
15       DB[key].readLV = ElemWiseMax(T.LV, DB[key].readLV);
16     if T writes DB[key] then
17       # T.LV is always no less than DB[key].writeLV
18       DB[key].writeLV = T.LV;
19     Release(key)
20   Asynchronously commit T if  $PLV \geq T.LV$  and all transactions in
21    $L_i$  with smaller LSNs have committed;
22 Function WriteLogBuffer(logRecord, recordSize)
23    $L_i.allocatedLSN[j] = L_i.logLSN$ ;
24    $lsn = AtomicFetchAndAdd(L_i.logLSN, recordSize)$ ;
25    $memcpy(L_i.logBuffer + lsn, logRecord, recordSize)$ ;
26    $L_i.filledLSN[j] = lsn + recordSize$ ;
27   return  $lsn + recordSize$ 

```

these partial orders enforces WAR and WAW dependencies. Namely, any previous transactions that ever read or wrote A will have an  $LV$  no greater than  $T_1.LV$ . ③  $T_1$  acquires a shared lock on B and then reads it. Then,  $T_1$  updates  $T_1.LV$  to be the element-wise maximum among B.*writeLV* and  $T_1.LV$ . This is to track RAW dependencies. Now  $T_1.LV = [\max(8, 4), \max(6, 7)] = [8, 7]$ . ④  $T_2$  wants to read A but has to wait for  $T_1$  to release the lock. ⑤ Similarly,  $T_3$  wants to write B but has to wait as well. ⑥ After  $T_1$  finishes,  $T_1$  writes its redo record and a copy of  $T_1.LV$  into the log buffer. After successfully writing to the buffer,  $T_1$  learns its LSN in Log 1 is 16. Then,  $T_1$  updates the first dimension of  $T_1.LV$  to be 16, resulting in  $T_1.LV = [16, 7]$ . ⑦ For each item  $T_1$  accessed,  $T_1$  updates the *readLV* (or *writeLV*) accordingly.  $T_1$  updates A.*writeLV* =  $ElemWiseMax(A.writeLV, T_1.LV) = T_1.LV = [16, 7]$ , and B.*readLV* =  $ElemWiseMax(B.readLV, T_1.LV) = [16, 11]$ . Then,  $T_1$  releases the locks. After this,  $T_1$  waits for itself and all the transactions it depends on to become persistent, equivalently,  $PLV \geq T_1.LV$ . The workers can process other transactions, and periodically check if  $T_1$  should be marked as committed. ⑧  $T_2$  acquires the shared lock on A.  $T_2$  then updates  $T_2.LV = ElemWiseMax(T_2.LV, A.writeLV) = [16, 7]$ . This update enforces the partial order that  $T_1.LV \leq T_2.LV$  because  $T_2$  depends on  $T_1$ . Since  $T_2$  is read-only, it does not create a log record. It also enters the asynchronous commit by waiting for  $PLV \geq T_2.LV$ . ⑨  $T_3$  acquires an exclusive lock on B and updates  $T_3.LV = ElemWiseMax(T_3.LV, B.readLV, B.writeLV) = [16, 11]$ . The fact that  $T_3$  depends on  $T_1$  reflects on





**Figure 3: Worker Thread Example** – Three transactions ( $T_1$ ,  $T_2$ , and  $T_3$ ) are accessing two objects A and B. Transactions are logged to two files. The diagram is drawn in the time order with the axis on the left.

#### Algorithm 2: Log Manager Thread $L_i$

```

1 readyLSN =  $L_i$ .logLSN;
2 foreach worker thread  $j$  that maps to  $L_i$  do
   # We assume allocatedLSN[j] and filledLSN[j] are fetched
   # together atomically;
3   if allocatedLSN[j]  $\geq$  filledLSN[j] then
4     readyLSN = min(readyLSN, allocatedLSN[j])
5 flush the buffer up to readyLSN;
6 PLV[i] = readyLSN;

```

$T_3.LV \geq T_1.LV$ . ⑩ The logging threads have flushed all transactions before  $T_1.LV = T_2.LV = [16, 7]$  and updated PLV. Observing  $PLV \geq [16, 7]$ , Taurus marks  $T_1$  and  $T_2$  as committed. ⑪  $T_3$  writes its redo record and a copy of  $T_3.LV$  to the buffer of Log 2, and gets its LSN as 21.  $T_3.LV$  increases to  $[16, 21]$ . ⑫  $T_3$  sets B.writeLV to  $[16, 21]$  and releases the lock. ⑬ When PLV achieves  $T_3.LV = [16, 21]$ , Taurus commits  $T_3$ .

**Log Manager Threads:** We use a dedicated thread serving as the log manager for each log file. The main job of the log manager is to flush the contents in the log buffer into the file on disk. It periodically invokes Alg. 2 when a timeout period has passed or when the buffer is half full, whichever happens first. The algorithm identifies up to which point the DBMS can flush to the disk so that it does not flush data that active transactions are still processing.

Taurus uses two arrays, *allocatedLSN* and *filledLSN*, to achieve this goal. *readyLSN* is the log buffer position up to which the DBMS can safely flush; its initial value is *logLSN*[ $i$ ] (Line 1). For each worker thread  $j$  that belongs to  $L_i$ , if *allocatedLSN*[ $j$ ]  $\geq$  *filledLSN*[ $j$ ], then the transaction in thread  $j$  is filling the log buffer at a position after *allocatedLSN*[ $j$ ] (Alg. 1, Line 20 and Line 23), so *readyLSN* should not be greater than *allocatedLSN*[ $j$ ]. Otherwise, no transaction in worker  $j$  is filling the log buffer, so *readyLSN* is not changed (Lines 2–4). Lastly, the log manager flushes the buffer to the disk up to *readyLSN* and updates *PLV*[ $i$ ] (Lines 5–6).

The frequency that the DBMS flushes log records to disk is based on the performance profile of the storage devices. Although each flush might enable a number of transactions to commit, transactions in the same log file still commit in a sequential order. This removes

#### Algorithm 3: Log Manager Recovery for Thread $L_i$ .

```

1 while  $T = L_i.DecodeNext()$  and  $T.LV \leq ELV$  do
2   pool.Enqueue( $T$ );
3   pool.maxLSN =  $T.LSN$ ;

```

ambiguity of transaction dependency during recovery. Sequential committing will not affect scalability because ELR prevents transactions waiting for log record duration or sequential committing from being on the critical path.

### 3.3 Recovery Operations

Taurus' recovery algorithm replays transactions following the partial orders between their LVs, which is sufficient to respect all the data dependencies. This is equivalent to performing topological sorting in parallel on a dependency graph. Each log manager thread reads log records from a file, and the worker threads recover transactions by re-applying the log records.

**Data Structures:** The recovery process contains the following:

- *L.pool* – For each log manager, *pool* is a queue containing transactions that are read from the log but not recovered.
- *L.maxLSN* – For each log manager, *maxLSN* is the LSN of the latest transaction that has been read from the log file.
- *Global.RLV* – *RLV* is a vector of length  $n$  ( $n$  is the number of log managers). An element  $RLV_i$  means that all transactions mapping to  $L_i$  with  $LSN \leq RLV_i$  have been successfully recovered. Therefore, a transaction  $T$  can start its recovery if  $T.LV \leq RLV$ , at which point all transactions that  $T$  depends on have been recovered. Initially, *RLV* is a vector of zeroes.
- *Global.ELV* – *ELV* is a vector of length  $n$ . An element  $ELV_i$  is the number of bytes in *Log i*. The DBMS uses this vector to determine if a transaction committed before the crash. Before the recovery starts, Taurus fetches the sizes of the log files to initialize *ELV*, namely,  $ELV[i]$  is the size of *Log i*.

**Log Manager Threads:** In Alg. 3, the thread reads the log file and decodes records into transactions (Line 1). For a transaction  $T$ , if  $T.LV \leq ELV$ , then  $T$  committed before the crash and is therefore considered for recovery; otherwise,  $T$  and transactions after it are ignored for recovery. A transaction is enqueued into the tail of *pool* and the value of *maxLSN* is updated to be the LSN of  $T$  (Lines 2–3). It is important that the thread updates *maxLSN* after it executes *Enqueue*, otherwise the DBMS may recover transactions in an incorrect order. If the pool is empty after the DBMS updates *maxLSN* but before it enqueues  $T$ , then it sets  $RLV[i] = T.LSN$  to indicate that  $T$  is recovered; this prevents another worker from recovering a transaction that depends on  $T$  before  $T$  is recovered.

**Worker Threads:** In Alg. 4, the worker threads keep executing until the log manager finishes decoding all the transactions and the pool is empty. A worker thread tries to get a transaction  $T$  from *pool* such that  $T.LV \leq RLV$  (Line 2). Then, the worker thread recovers  $T$  (Line 3). For data logging, the data elements in the log record are copied to the database; for command logging, the transaction is re-executed. During the re-execution, no concurrency control algorithm is needed, since Taurus guarantees no conflicts during recovery. Then,  $RLV[i]$  is updated (Lines 4-7). If *pool* is empty,

**Algorithm 4: Worker Recovery Thread**

```

1 while not IsRecoveryDone() do
  # FetchNext atomically dequeues a transaction T such that
  # T.LV ≤ RLV;
2   T = pool.FetchNext(RLV);
3   Recover(T);
4   if pool is empty then # Atomic Section
5     RLV[i] = Max(RLV[i], pool.maxLSN);
6   else
7     RLV[i] = Max(RLV[i], pool.head.LSN - 1)

```

then the thread sets  $RLV[i]$  to  $pool.maxLSN$ , the largest LSN of any transaction added to  $pool$ , if it is larger; otherwise,  $RLV[i]$  is set to one less than the first transaction’s LSN, indicating that the previous transaction has been recovered but not the one blocking the head of  $pool$ . In the pseudo-code, the code for  $RLV$  update is protected with an atomic section for correctness. We use a lock-free design to avoid this critical section in our implementation.

The  $pool$  data structure described above can become a potential scalability bottleneck if a large number of workers are mapped to a single log manager. There are additional optimizations that address this issue. For example, we partition each  $pool$  into multiple queues. We also split  $RLV$  into local copies and add delegations to reduce false sharing in CPU caches.

### 3.4 Supporting Index Operations

Although our discussion has focused on *read* and *update* operations, Taurus can also support *scan*, *insert*, and *delete* operations with an additional index locking protocol.

For a range scan, the transaction (atomically) fetches a shared lock on each of the result rows using the *Lock* function in Alg. 1. When the transaction commits, it goes through the *Commit* function and update the  $readLV$ ’s of the rows. To avoid phantoms, the transaction performs the same scan again before releasing the locks in *Commit* function. If the result rows are different, some other transactions have inserted or deleted rows within the scan range, we abort the transaction. This scan-twice trick is from Silo [35]. We notice that, assuming 2PL, the transaction only needs to record the number of rows returned. During the second scan, it’s guaranteed that the rows in the previous scan still exist because shared locks are held by the transaction. Therefore, if the row count is still the same, the result rows are not changed.

If a transaction  $T$  inserts a row with primary key  $key$ , it initializes  $DB[key].readLV$  and  $DB[key].writeLV$  to be 0. Because the index for  $DB[key]$  is not updated yet, other transactions will not see the new row. In *Commit* function after  $T$  releases the locks, it updates  $DB[key].writeLV = T.LV$ . Finally,  $T$  inserts  $key$  into the index.

When a transaction  $T$  deletes a row with primary key  $key$ , it first grabs an exclusive lock of the row. And updates  $T.LV = ElemWiseMax(T.LV, DB[key].readLV, DB[key].writeLV)$ . Any other transaction trying to access this row will abort due to lock conflicts. In the *Commit* function before  $T$  releases the locks, it removes  $key$  from the index.

### 3.5 Limitations of Taurus

We now discuss the limitations of Taurus’s design and potential ways to mitigate them.

One potential problem is that the size of  $LV$  is proportional to the number of log managers. For a large number of log managers, the computation and storage overhead of  $LV$  will increase. In contrast, serial logging maintains a single LSN and therefore avoids this problem. Although we believe most DBMSs will use a relatively small number of log files and thus this overhead is acceptable, Taurus can also leverage  $LV$  compression (Sec. 4.1) and SIMD instructions (Sec. 5.6) to partially resolve this issue. If necessary, a dependency-aware transaction-to-log mapping mechanism can also potentially reduce inter-log dependencies.

Another limitation of Taurus is the amount of parallelism during recovery for workloads with high contention. For these workloads, the inherent recovery parallelism can be lower than the number of log managers. During recovery, a large number of inter-log dependencies will exist. In Taurus, the dependencies propagate through  $RLV$  (Alg. 4), which leads to inter-thread communication, incurring relatively long latency between the recovery of dependent transactions. In contrast, a serial recovery scheme has no delay between consecutive transactions and, therefore, may deliver better performance. To address this, when the contention is high, Taurus will degrade to using serial recovery. Specifically, a single worker recovers all the transactions sequentially. The worker checks every  $pool$  of log managers and recovers the transaction that satisfies  $T.LV \leq RLV$ ; this approach incurs no delay between two consecutive transactions. We evaluate this aspect in Sec. 5.6.

During recovery, if the pool size is large and contention is high, workers might need to scan the whole pool to find the next transaction that is ready to be recovered. Heuristic optimizations like zig-zag scans could help. We defer the problem of developing a data structure specialized for Taurus recovery to future work.

## 4 Optimizations and Extensions

We now discuss optimizations to reduce Taurus’  $LV$  storage overhead and computational overhead, and two extensions to support Optimistic Concurrency Control (OCC) and MVCC.

### 4.1 Optimization: LV Compression

The design of Taurus as described in Sec. 3 has two issues: (1) the DBMS stores  $readLV$  and  $writeLV$  for every tuple, which changes the data layout and incurs extra storage overhead; (2) the transaction’s  $LV$  is stored for each log record, which can significantly increase the log size especially for command logging where each log record is relatively small. We describe optimizations that address these problems.

**Tuple LV Compression:** To reduce a tuple’s  $LV$  storage, we observe that keeping  $LV$ s for tuples that were accessed a long time ago is unnecessary. The  $LV$ s in these tuples are too small to affect active transactions. This optimization thus stores  $LV$ s only for active tuples in the lock table. For these tuples, transactions operate on their  $LV$ s following Alg. 1. If the DBMS inserts a tuple into the lock table, then the algorithm assigns its  $readLV$  and  $writeLV$  to be the current  $PLV$ . The system can evict a tuple from the lock table if

**Algorithm 5: LV Compression for Log Records** – Each log manager  $L_i$  periodically calls *FlushPLV*. The worker threads mapped to  $L_i$  call *Compress* and *Decompress*.

```

1 Function FlushPLV()
2    $currentPLV = Global.PLV$ ;
3    $logBuffer.append(currentPLV)$ ;
4    $LPLV = currentPLV$ ;
5 Function Compress( $LV$ )
6    $compressedLV = LV$ ;
7   foreach  $LV[j] \in LV$  do
8     if  $LV[j] \leq L_i.LPLV[j]$  then
9        $compressedLV[j] = NaN$ ;
10  return  $compressedLV$ ;
11 Function Decompress( $compressedLV$ )
12   $LV = compressedLV$ ;
13  foreach  $LV[j] \in LV$  do
14    if  $LV[j] = NaN$  then
15       $LV[j] = L_i.LPLV[j]$ ;
16  return  $LV$ ;

```

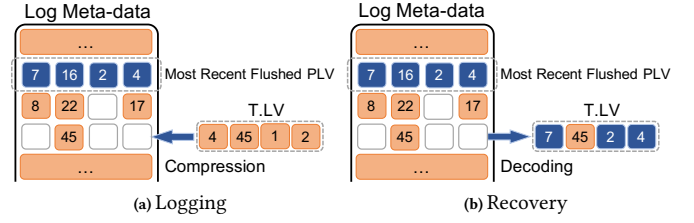
no transactions hold locks on it and both its *readLV* and *writeLV* are not greater than the current *PLV*.

For the tuples previously evicted from the lock table and later inserted back, the optimization increases the *readLV* and *writeLV* of these tuples and also the *LVs* of transactions accessing them. This modification makes transactions depend on more transactions than before. To make the trade-off between higher compression ratio and fewer artificial dependencies, we introduce a new parameter  $\delta$  and evict a tuple from the lock table only if  $\forall i, PLV[i] - LV[i] \geq \delta$  is true for both *readLV* and *writeLV*. Accordingly, a newly inserted tuple will have  $readLV[i] = writeLV[i] = PLV[i] - \delta$ . Larger  $\delta$  means fewer artificial dependencies, but more tuples will stay in the lock table waiting for eviction, and vice versa.

**Log Record LV Compression:** We next address the second issue that each log record must store the *LV* of the transaction. We propose an optimization where each log record stores only a few but not all dimensions of a transaction’s *LV*. The motivating insight is that for workloads with low to medium contention, most dimensions of a log record’s *LV* are too small to be interesting. For example, suppose that a transaction  $T$  depends on a committed transaction  $T'$ . It is not critical to remember precisely which  $T'$  that  $T$  depends on, but only that  $T$  depends on some transaction that happened before a specific point in time. Therefore, we can set anchor points (in the form of *LVs*) into each log such that, if  $T$  depends on only transactions before an anchor point, it stores the anchor point instead of the detailed *LV*.

In Alg. 5, we introduce a variable *LPLV* as the anchor point. *L.LPLV* is an LSN Vector that is maintained by each log manager  $L$ . It keeps a copy of the most recent *PLV* written into  $L$ ’s log buffer. Periodically, the log manager appends *PLV* into the log buffer and updates *L.LPLV* (Lines 1–4).

To compress a transaction  $T$ ’s *LV*, we check for every dimension if  $T.LV$  is no greater than  $L_i.LPLV$ . If this is true for dimension  $j$ , namely,  $T.LV[j] \leq L_i.LPLV[j]$ , we can artificially increase  $T.LV[j]$  to  $L_i.LPLV[j]$ . Since  $L_i.LPLV$  is already in the buffer, the system



**Figure 4: LV Compression** – Example of Taurus’s compression method.

no longer needs to store  $T.LV[j]$  (Lines 6–9). During recovery, the DBMS performs the opposite operation; if the  $j$ -th dimension of an *LV* was compressed, it replaces it with the value of *LPLV*[ $j$ ] (Lines 12–15). If it reads an anchor from the log, it updates *LPLV*.

Fig. 4 shows an example of *LV* compression. In Fig. 4a, transaction  $T$ ’s *LV* = [4, 45, 1, 2] is written to the log. The system compares it against *LPLV* and finds that  $T.LV$  has only one dimension (the 2nd dimension with value 45) greater than *LPLV*. Only the 2nd dimension is written into the log. During recovery, Fig. 4b shows that Taurus fills in the blanks with the most recently seen anchor, *LPLV* = [7, 16, 2, 4]. The compressed *LV* is decoded into [7, 45, 2, 4]. Note that the 1<sup>st</sup>, 3<sup>rd</sup>, and 4<sup>th</sup> dimension of the decompressed *LV* are greater than the original  $T.LV$ .

The frequency of *LPLV* flushing makes a trade-off between parallelism in recovery and *LV* compression ratio. When the frequency is high, a dimension of *LV* is smaller than *LPLV* and thus it enables better compression, but some amount of recovery parallelism is sacrificed since the decompressed *LVs* have larger values.

## 4.2 Optimization: Vectorization

The logging overhead mainly consists of four parts: (1) the overhead introduced by Taurus where we calculate *LVs* and move them around; (2) the overhead of creating the log records and writing them to the in-memory log buffer; (3) for lock-based concurrency control algorithms, the extra latency caused by (1) and (2) will result in extra lock contention; (4) the time cost in persisting the log records to the disk. Among them, (4) is moved off the critical path by ELR; (2) and (3) are shared by essentially all the write-ahead logging algorithms. These overheads will not block the DBMS from scaling up. Overheads (1) and (2) are linear in the number of total transactions executed. Overhead (1) is also related to the number of log files. If the system is writing to many log files and the transactions have a short execution time, it is up to 13.8% of the total execution time if implemented naively. We can exploit the data parallelism in the LSN Vector as the values within a single vector are processed independently. Modern CPUs provide SIMD extensions that allow the DBMS to process multiple vector elements items in a single instruction. For example, the instruction `_mm512_max_epu32` can compute the element-wise maximum of two vectors of 16 32-bit integers. In Sec. 5.6, we show that switching to vectorized operations reduces Taurus’ overhead by 89.5%.

## 4.3 Extension: Support for OCC

Our overview of Taurus thus far assumes that the DBMS uses 2PL. Taurus is also compatible with other schemes. We next discuss how Taurus can support Optimistic Concurrency Control (OCC) [24].



**Algorithm 6: OCC Logging for Worker Threads**


---

```

1 Function Access(key, T)
2   value, readLV, writeLV = load(key) # load atomically;
3   T.LV = ElemWiseMax(T.LV, writeLV);
4   return value
5 Function Commit(T)
6   for key  $\in$  sorted(T.writeSet) do
7     DB[key].lock();
8   for key  $\in$  T.readSet do
9     foreach dimension  $i$  of LV do
10      if DB[key].readLV[i] < T.LV[i] then # Atomic
11        DB[key].readLV[i] = T.LV[i]
12   if not ValidateSuccess() then
13     Abort(T);
14   Create log record and write to log buffer similar to Lines 8–11 in
15   Alg. 1;
16   for key  $\in$  T.writeSet do
17     DB[key].writeLV = ElemWiseMax(DB[key].writeLV, T.LV);
18     DB[key].release();
19   Asynchronously commit  $T$  if  $PLV \geq T.LV$  and all transactions in
20    $L_i$  with smaller LSNs have committed;

```

---

Alg. 6 shows the protocol for a worker thread. Different from a 2PL protocol (Alg. 1), an OCC transaction calls *Access* when accessing a tuple and *Commit* after finishing execution. The *readSet* and *writeSet* are maintained by the *read/write* functions in the conventional OCC algorithm, from which *Access* is called. In the *Access* function, the transaction atomically reads the value, *readLV*, *writeLV*, and potentially other auxiliary data. Commonly seen in OCC algorithms, the *ValidateSuccess* function returns true if the values in the *readSet* are not modified by other transactions. The atomicity is guaranteed through a latch, or by reading a version number twice before and after reading the value [35].

For high concurrency, we choose a reader-lock-free design of the *Commit* function. The transaction first locks all the tuples in the *writeSet* (Lines 6–7). Before validating the *readSet* (Line 12), it updates the *readLV* of tuples in the *readSet* one dimension at a time (Lines 9–11). Each update happens atomically using compare-and-swap instructions. This is necessary because the data item might appear in the *readSet* of multiple transactions, and concurrent updates of *readLV* might cause loss of data. The reason that the *readLV* extension must occur before the validation is to enforce write-after-read dependencies. To see a failure example, consider a transaction  $T_1$  modifying the data after  $T_2$ 's validation but before  $T_2$ 's updates on *readLV*. Then, it is possible that  $T_1$  does not observe the latest *readLV* and therefore fails to capture the write-after-read dependency to  $T_2$ . Note that updating *readLV* before the validation might result in extra non-existing dependencies (i.e., *LV*s larger than necessary) where the transaction aborts later in the validation but has already updated the *readLV* of some data tuples. Such aborts only affect performance but not correctness. The design of log managers stays the same as in Alg. 2.

**4.4 Extension: Multi-Versioning**

We next discuss how Taurus works with MVCC. We assume the recovery process also uses multi-versions. Otherwise, the DBMS has to reorder the transactions either by changing already persistent data or appending extra information. Concurrency control algorithms based on logical timestamps allow physically late transactions to access versions early and commit transactions logically early. The DBMS, however, creates log records and flushes them in the physical time order. Solving the decoupled order requires extra design. Allowing multi-versions in the recovery process relaxes the decoupling by allowing physically late transactions to commit logically early in the recovery. This assumption frees Taurus from tracking the WAR dependencies because the read operation can still fetch the correct historic version even after the tuple has been modified. Therefore, Taurus only needs to track WAW and RAW dependencies. Different from Sec. 3.1, Taurus for MVCC only adds a single metadata field for the data versions, the LSN Vector *LV*. Our discussion is based on the MVCC scheme [25] used in Hekaton [9]. The algorithm adds three extra fields to the data version tuples, namely, Begin Timestamp, End Timestamp, and a hash pointer.

Whenever a transaction reads a data version  $v$ , the transaction updates *T.LV* to be *ElemWiseMax(T.LV, v.LV)* to catch RAW dependencies. When a transaction updates the data by adding a new data version  $v$  after the old version  $u$  during normal processing phase, it first updates the timestamps as in MVCC, then it updates *T.LV* to be *ElemWiseMax(T.LV, u.LV)*, and  $v.LV$  to be empty.

In the postprocessing phase, if the transaction  $T$  commits, before it replaces its transaction ID with its end timestamp, it iterates data versions in the *writeSet*. For a data version  $v$  in the *writeSet*, it replaces  $v.LV$  to be *T.LV*. The log records of  $T$  contains *T.LV* and the commit timestamp of  $T$ . The former identifies whether  $T$  should recover and the recovery order, and the latter determines the visible version when reading the data as well as the logical timestamp of the new versions when writing the data.

During recovery, Alg. 3 and Alg. 4 are executed. Only the visible version is returned for read operations. Whenever a write happens, the transaction writes a new version with the commit timestamp. Different from MVCC, transactions no longer acquire locks during recovery because Taurus guarantees no conflicts will occur. Without Taurus, the log records described in [25] contain only the payload and the logical timestamps, enforcing a total order among transactions. Taurus exploits the parallelism to recover non-conflicting transactions in parallel.

**5 Evaluation**

We implemented Taurus in the DBx1000 in-memory DBMS [1] to evaluate its performance. We include both the 2PL and OCC variants of Taurus in the evaluation. We evaluate the DBMS on three storage devices: (2) NVMe SSDs, (2) hard drives (HDDs), and (3) Persistent Memory (PM) simulated by a RAM disk. The performance profiles of these devices highlight different properties of the logging algorithms. As the mainstream fast storage, NVMe SSDs provide a high bandwidth, affording insights of the performance in production. HDDs have limited bandwidth, which is better for command logging. The cutting-edge PM largely eliminates disk bandwidth restrictions and exposes CPU and memory overheads.

We compare Taurus to the following protocols all in DBx1000:

**No Logging:** The DBMS has all logging functionalities disabled. It does not incur any logging-related overhead and therefore serves as a performance upper bound.

**Serial Logging:** This is our baseline implementation that uses a single disk and supports both data logging and command logging. For data logging, the DBMS saturates the 160 MB/s bandwidth. With command logging, the DBMS generates a smaller log and its performance is limited by the atomic increment of the central LSN.

**Serial Logging with RAID-0 Setup:** This is the same configuration as Serial Logging, except that it uses a RAID-0 array across the eight disks using Linux’s software RAID driver.

**Plover:** This is a parallel data logging scheme that partitions log records based on data accesses [44]. It uses per-log sequence numbers to enforce a total order among transactions. Each transaction generates multiple log entities.

**Silo-R:** Lastly, we also implemented the parallel logging scheme from Silo [35, 43]. Silo uses a variant OCC that commits transactions in epochs. The DBMS logs transactions in batches that are processed by multiple threads in parallel. Silo-R only supports data logging because the system does not track write-after-read dependencies.

## 5.1 Workloads

We first describe the benchmarks used in the evaluation:

**Yahoo! Cloud Serving Benchmark (YCSB):** This benchmark simulates the workload pattern of cloud-based OLTP systems [7]. In our experiments, we simulate a DBMS with a single table. Each data row has 10 fields and each field contains 100 bytes. We evaluate two databases with 10 GB and 500 GB of data. We build a single index for the table. The access pattern of transactions visiting the rows follows a Zipfian distribution; unless otherwise stated, we set the distribution parameter to 0.6 to simulate moderate contention. By default, each transaction accesses two tuples and each access has a 50% chance to be a read operation and a 50% chance to be a write operation. We will perform sensitivity studies regarding these workload parameters in Sec. 5.6. The size of each transaction’s command log record is smaller than that of its data log records.

**TPC-C:** This is the standard OLTP benchmark that simulates a wholesale company operating on warehouses [34]. The database has nine tables covering a variety of necessary information and transactions are performing daily order-processing business. We simulate two (Payment and New-Order) out of the five transaction types in TPC-C as around 90% of the default TPC-C mix consists of these two types of transactions. When Taurus is running in command logging mode, each transaction log record consists of the input parameters to the stored procedure. The workload is logically partitioned by warehouses. We use 80 warehouses in the evaluation. We evaluate the full TPC-C workload in Sec. 5.5.

The choices of the benchmarks provide a comprehensive evaluation of Taurus and baselines. YCSB, TPC-C Payment, and TPC-C New-Order represent short transactions with moderate contention, short transactions with low contention, and long transactions.

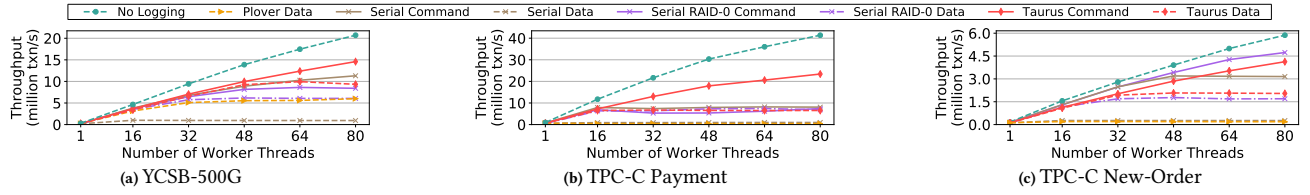
## 5.2 Performances with NVMe SSDs

We run the DBMS on an Amazon EC2 `i3en.metal` instance with two Intel Xeon 8175M CPUs (24 cores per CPU) with hyperthreading (96 virtual cores in total). The server has eight NVMe SSDs. Each device provides around 2 GB/s bandwidth and in total the server has theoretically 16 GB/s I/O bandwidth. We run the DBMS with at most 80 worker threads and 16 log manager threads. Every disk contains two log files to better exploit the bandwidth.

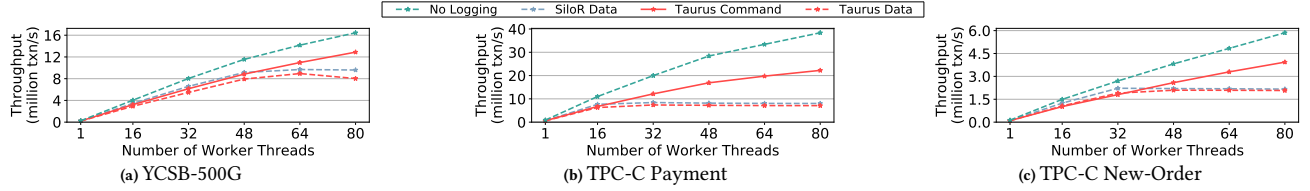
**Logging Performance:** Our first experiment evaluates the runtime performance of Taurus by measuring the throughput when the number of worker threads changes. We test the logging protocols with YCSB-500G and TPC-C benchmarks. We measure the throughput by the number of transactions committed by the worker threads per second. We keep the 2PL and OCC results separate to avoid comparisons based on the concurrency control algorithm performance. We show the 2PL results in Fig. 5 and the OCC results in Fig. 6. The x-axes are the number of worker threads (excluding the log managers), and the y-axes are the execution throughput.

Fig. 5a presents the logging performance for the YCSB-500G benchmark. Taurus with command logging scales linearly, while Taurus with data logging plateaus after 48 threads because it is bounded by the I/O of 16 dedicated writers. The serial command baseline also reaches a high throughput due to the succinctness of the command logging. It grows slower after 48 threads. This is not because of the disk bandwidth because it achieves similar performance with the RAID-0 disk array. It is instead because every transaction that spans multiple threads increments the shared LSN; this leads to excessive cache coherence traffic that inhibits scalability [35]. Taurus command logging is more scalable as the number of worker threads increases because each log manager maintains a separate LSN. Serial data saturates the single disk’s bandwidth. Similar to Taurus, Plover writes records across multiple files. For every transaction, it generates a log record for each accessed partition, and accesses the per-log LSN to generate a global LSN for the transaction. The DBMS then uses this global LSN to update the per-log sequence numbers. These updates are atomic to prevent data races. Plover is limited by the contention of the local counters. Taurus with command logging is up to 2.4× faster than Plover.

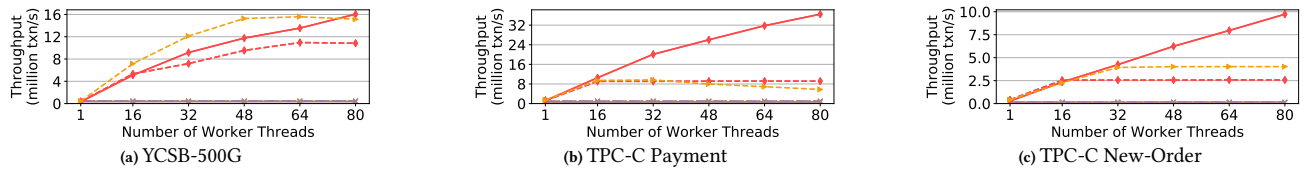
Fig. 5b shows the performance for the short and low-contented Payment transactions. These results are similar to YCSB. All the logging baselines incur a significant overhead compared to No Logging. The gap between No Logging and Taurus reflects the overheads discussed in Sec. 4.2. The LV maintenance in Taurus only costs 1.6% of the running time. Taurus with command logging has the best performance. Plover suffers from the increased data accesses, causing the worker threads to compete for one of the few latches on the local sequence numbers, essentially downgrading to a single stream logging. Fig. 5c shows the comparison for the TPC-C New-Order transactions. These transactions access a larger number of tuples (~30 tuples per invocation) than the previous workloads. The overall throughput is lower, making it difficult for the DBMS to hit the LSN allocation bottleneck. Therefore, serial command logging scales well. The gap between serial command with RAID-0 and Taurus command corresponds to LV-related overheads. Taurus with command logging shows advantages when the number of workers is adequate. We project that the serial command logging



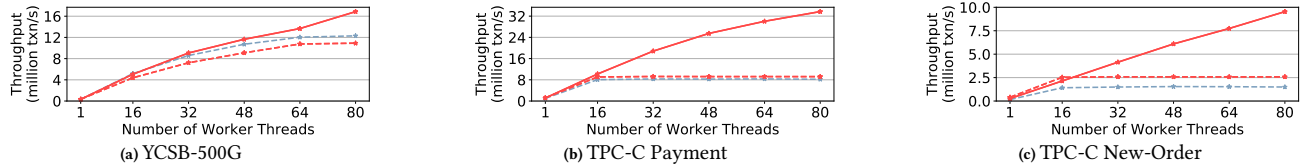
**Figure 5: Logging Performance (2PL)** – Performance comparison on YCSB-500G, TPC-C Payment, and TPC-C New-Order on NVMe drives.



**Figure 6: Logging Performance (OCC)** – Performance comparison on YCSB-500G, TPC-C Payment, and TPC-C New-Order on NVMe drives.



**Figure 7: Recovery Performance (2PL)** – Performance comparison on YCSB-500G, TPC-C Payment, and TPC-C New-Order on NVMe drives.



**Figure 8: Recovery Performance (OCC)** – Performance comparison on YCSB-500G, TPC-C Payment, and TPC-C New-Order on NVMe drives.

will plateau reaching the cache traffic limit when there are more than 120 workers whereas Taurus should still scale. Similar to Payment transactions, Plover is bounded by the contention.

Fig. 6 shows the comparison between the OCC variant of Taurus and Silo-R. The No Logging baseline also uses the OCC algorithm to keep comparison fair. For all the benchmarks we observe that both Silo-R and Taurus data logging plateaus at a similar level, saturating the disk bandwidth. Before they saturate the bandwidth, Silo-R performs slightly better than Taurus because it does not need to track LSN Vectors. However, Silo-R cannot track RAW dependencies, so it is incompatible with command logging. Taurus command logging, benefiting from the conciseness of the log records, outperforms Silo-R in every benchmark, by up to 2.8 $\times$ .

**Recovery Performance:** We next evaluate the DBMS’s recovery time for all the protocols. We use the log files generated by 80 worker threads for better recovery parallelism. These files are large enough for steady performance measurements and are stored in uncompressed bytes across the disks with I/O caches cleaned.

Fig. 7a shows the recovery performance on YCSB-500G. Plover outperforms Taurus below 80 threads because it does not need to resolve dependencies. Every Plover log file corresponds to a partition that contains totally ordered records, which is sufficient to recover transactions independently. Plover saturates the disk’s 16 GB/s bandwidth after 48 threads and plateaus. Taurus command scales linearly and exceeds Plover at 80 threads. The serial baselines, regardless of data or command logging, with a RAID-0 setup or

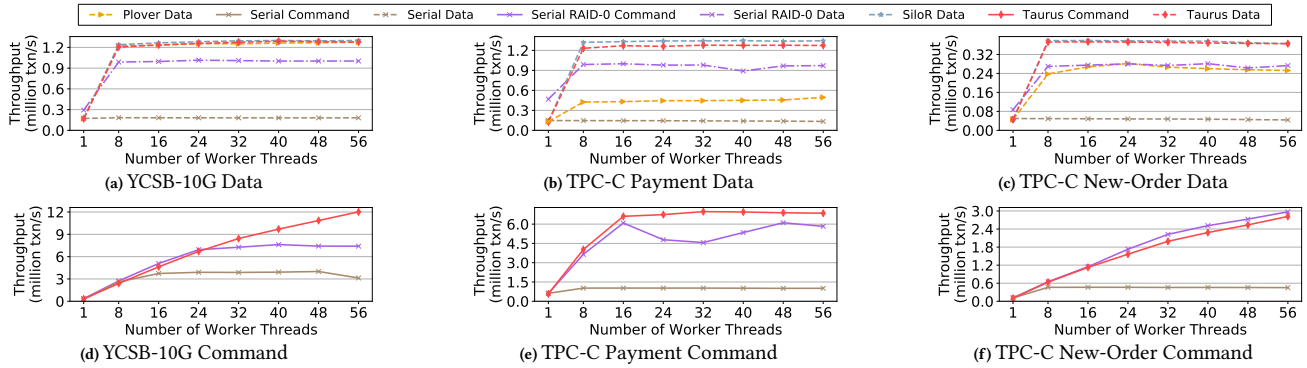
not, are limited by the total sequence order of transactions. Taurus recovery is up to 42.6 $\times$  faster than the serial baselines.

The recovery performance of TPC-C Payment is in Fig. 7b. Both Plover and Taurus data logging hit the I/O bottleneck quickly, while Taurus command logging scales linearly. Fig. 10f shows the comparison for TPC-C New-Order. Taurus command scales well and outperforms Plover by up to 2.4 $\times$ . The gap between Plover and Taurus data logging corresponds to dependency resolution and the resulting memory overhead. Taurus command is slower than Taurus data at 16 threads due to the cost of re-running the transactions.

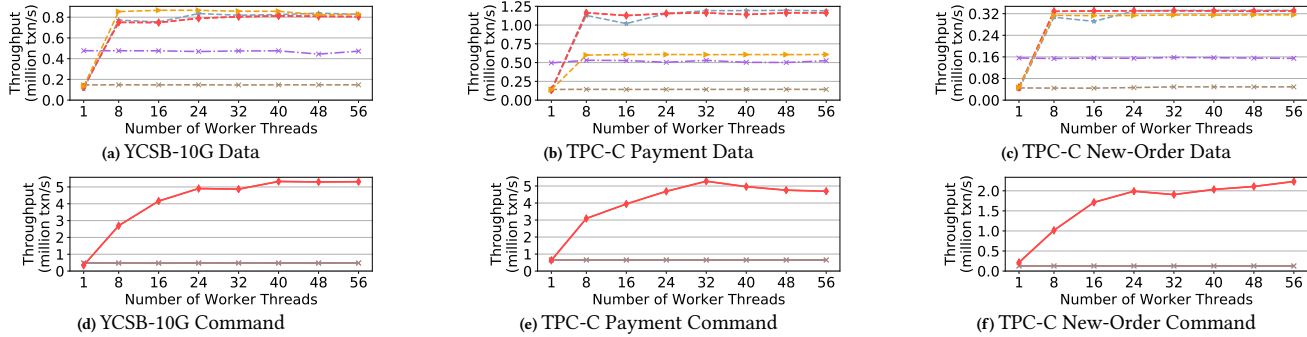
Fig. 8 presents the comparisons for the OCC baselines. Similar to Plover, Silo-R requires data logging and therefore falls behind Taurus command logging in all three benchmarks. But Silo-R does not require dependency resolution so it outperforms Taurus with data logging when the number of transactions is large. Silo-R uses latches when transactions update tuples to ensure that they only perform updates with a higher version number. This overhead is more significant when the transactions are long. Taurus command logging outperforms Silo-R by up to 9.7 $\times$ .

### 5.3 Performance with Hard Disks

To better understand the performance of baselines with limited bandwidth, we performed the evaluation on an Amazon EC2 h1. 16xlarge machine with eight HDD drives. Each disk provides around 160 MB/s bandwidth and in total the server has 1.3 GB/s I/O bandwidth. Because the server only has 256 GB memory, we use YCSB-10G. The



**Figure 9: Data and Command Logging Performance** – Performance comparison on YCSB-10G, TPC-C Payment, and TPC-C New-Order on HDDs.



**Figure 10: Data and Command Recovery Performance** – Performance comparison on YCSB-10G, TPC-C Payment, and TPC-C New-Order on HDDs.

data logging and command logging baselines differ in absolute throughput on HDDs, so we present them separately. Silo-R is bound by the disk bandwidth often, and the difference in concurrency control does not contribute to the relative order. Therefore, we display the results for Silo-R and 2PL baselines together.

**Logging Performance:** Fig. 9a shows the logging performance of data logging baselines for YCSB-10G. We observe that serial data saturates the bandwidth of a single disk quickly. Taurus Data achieves 7.1× higher throughput than serial data as it writes to eight disks in parallel. Serial data logging on RAID-0 delivers similar performance since the bandwidth of the disk array is 8× greater. Silo-R and Plover also write the log across eight disks uniformly, thereby achieving similar performance to Taurus. In Figs. 9b and 9c, we also observe this pattern for the TPC-C transactions except that Plover plateaus because of the high contention.

Fig. 9d shows the command logging baselines for the YCSB benchmark. Serial command logging outperforms serial data logging, benefiting from the smaller log record sizes. Starting from 16 threads, its performance is limited by the bandwidth of a single disk. The serial command baseline on a RAID array plateaus after 24 threads, limited by the cache coherence traffic. Taurus with command logging is 9.2× faster than Silo-R and Plover. Fig. 9e shows the DBMS’s throughput for the TPC-C Payment transaction. Taurus plateaus after 16 threads, limited by the disk bandwidth, achieving 5.2× speedup over Silo-R. Serial command logging suffers from NUMA issues between 16 threads and 48 threads as the log buffer resides on a single socket. For the TPC-C New-Order workload in Fig. 9f, both serial command logging on the RAID-0 array and Taurus command logging have good scalability.

**Recovery Performance:** Fig. 10 presents the recovery performance on HDDs. The serial baselines are again limited by the transaction total order. For Taurus, we can see that the recovery performance of data logging plateaus after the number of worker threads exceeds 8. It is up to 1.7× faster than the serial data logging on a disk array. Taurus data logging achieves similar throughput as Silo-R, while Taurus command logging is up to 6.3× faster than Silo-R for recovery. Plover parallels Silo-R except for Payment, where the contention devolves Plover to single stream logging.

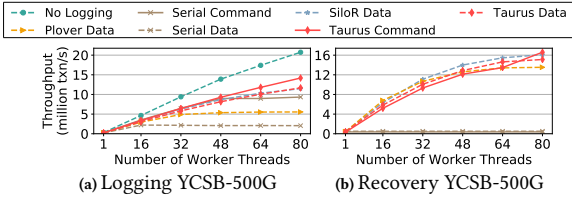
The peak performance of Taurus command logging and Taurus data logging are 11.3× and 5.5× faster than serial baselines for YCSB recovery. For TPC-C Payment in Fig. 10e, the DBMS achieves its peak recovery performance using Taurus command logging where it is 7.1× faster than the serial command logging baseline. The performance of Taurus command logging decreases when the number of workers increases beyond 24 because the parallelism is fully exploited and more threads will only incur more contention.

For TPC-C New-Order, the performance ratios between Taurus and the serial baselines are 17.5× and 6.7× for command logging (or data logging lifted by disk arrays) and data logging (without disk arrays), respectively. If the DBMS uses Taurus command logging protocol instead of its data logging protocol, then it improves the performance by 7.7×. This is up to 56.6× better than serial data logging. Databases with limited bandwidth can benefit from Taurus supporting command logging.

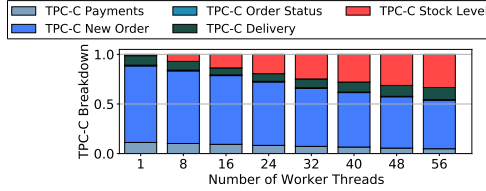
#### 5.4 Performance with PM (RAM Disk)

Since emerging Persistent Memory (PM) has higher I/O bandwidth than of SSDs and HDDs, we evaluated the performance of Taurus





**Figure 11: DRAM Performance** – Performance comparison on DRAM filesystems.



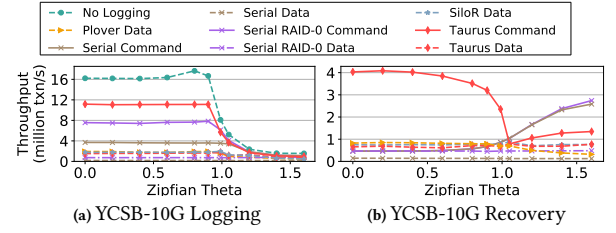
**Figure 12: TPC-C Full Mix Time Breakdown** – Time cost breakdown by different TPC-C transaction types.

on DRAM filesystems to simulate a PM environment. Every operation to this filesystem goes through the OS. This overhead is also shared in the architecture with a real PM. The PM incurs a higher latency ( $<1$  us for 99.99%) and has a bandwidth 3 – 13 $\times$  lower than DRAM [39]. We conjecture that Taurus command logging would perform relatively better than other baselines on a real PM because the bandwidth might become the bottleneck.

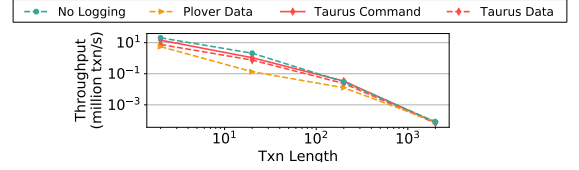
The results on the DRAM filesystem are shown in Fig. 11. The advantage of command logging over data logging is greatly reduced when the bandwidth is sufficient. Taurus command logging scales linearly, while the serial command logging is again restricted by the contention of the single counter. All the parallel algorithms scale well in recovery. Silo-R outperforms Taurus slightly because it does not have to resolve dependencies during recovery. We can infer that Taurus does not incur observable overhead that would preclude it from a PM-based DBMS.

## 5.5 TPC-C Full Mix

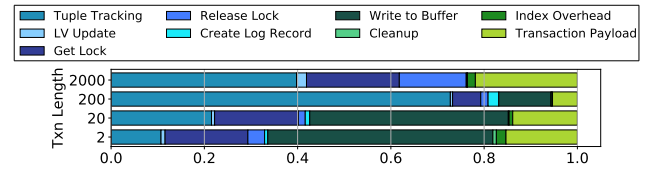
To demonstrate the generality of Taurus and to evaluate Taurus in a more realistic DBMS OLTP workload, we added the support for range scans, row insertions, and row deletions. We implement all the types of transactions from the TPC-C benchmark with the 2PL concurrency control algorithm. The full TPC-C mix consists of 45% New-Order, 43% Payment, 4% Order-Status, 4% Delivery, and 4% Stock-Level. Among them, Order-Status and Stock-Level are read-only transactions, and therefore Taurus does not create log records for them. Figure 16 shows the logging performance and recovery performance. We observe that, starting from 32 threads, the logging algorithms are limited by the workload parallelism since the no logging baseline plateaus at a similar level. Compared to the no logging baseline, the overhead caused by Taurus is around 11.7%. In recovery, the serial algorithms are again limited by the loss of parallelism. Taurus command logging outperforms the serial baselines by 12.8 $\times$ . Fig. 12 shows the time breakdown among all the five TPC-C transactions. Although Stock-Level transactions are read-only, they take a significant proportion of the total running time. This proportion increases with the number of threads because Stock-Level transactions perform massive read operations.



**Figure 13: Contention** – Zipfian Theta in YCSB.



**Figure 14: Transaction Impact** – We vary the number of tuples per transaction touches from 2 to 2000.



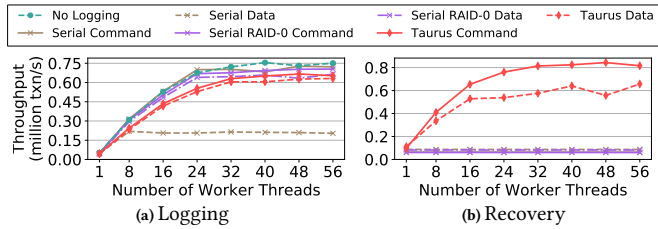
**Figure 15: Transaction Impact** – The percentages of the tuple tracking overhead for Taurus Data Logging.

## 5.6 Sensitivity Study

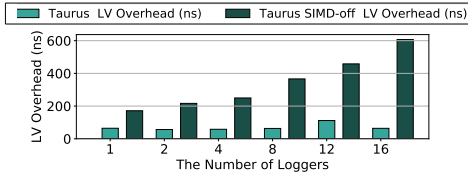
In this section we show that Taurus robustly provides relatively good performance even when various factors change.

**Contention:** We use the YCSB-10G benchmark on the h1.16xlarge server to study how the contention level impacts performance. We control the contention level by adjusting the  $\theta$  parameter of the Zipfian distribution. A higher  $\theta$  value corresponds to more contention. We include a baseline (No Logging) to provide an upper bound at different contention levels. Every baseline uses 56 worker threads.

Fig. 13 shows the DBMS’s throughput measurements when varying the Zipfian  $\theta$  parameter for the logging and recovery procedures. The gap between Taurus and No Logging is insensitive to data contention. When  $\theta$  is greater than 1.0, the performance of all logging schemes decreases due to the reduced parallelism in the workload. Fig. 13b shows the recovery measurements when the contention level increases. These results indicate different trends for serial algorithms and Taurus. For Taurus, the performance drops when  $\theta$  goes beyond 0.8 because of the inter-log dependency problem (Sec. 3.5): dependencies between transactions spanning different logs incur extra latency that hurts performance at high contention. In contrast, serial algorithms have low throughput at low contention, but their throughput increases with higher  $\theta$ . This is because higher data skew makes the working set fit in on-chip caches, resulting in a higher cache hit rate and thus better performance. Since the recovery proceeds sequentially, contention does not introduce data races, so it does not harm the performance of the serial baselines. When the contention level is high (i.e.,  $\theta > 1$ ), we run Taurus with serial recovery to avoid the high latency between dependent transactions. As shown in Fig. 13b, this configuration enables Taurus to achieve good performance under high contention.



**Figure 16: TPC-C Full Mix Performance (2PL)** – Performance comparison and scalability on TPC-C Full Mix.



**Figure 17: Vectorization** – The LV Update overhead (in nanosec).

**Transaction Impact** We evaluate how Taurus performs when every transaction touches a large number of tuples. We ran YCSB-500G on an Amazon EC2 *i3en.metal* instance and vary the number of tuples every transaction accesses from 2 to 2,000. Fig. 14 shows the throughput is inversely proportional with the transaction length. Fig. 15 shows the time breakdown of Taurus data logging for YCSB-10G. We can observe that when the number of tuples accessed per transaction increases from 2 to 200, the LV update overhead stays fixed at 0.6%, while the tuple tracking overhead of the 2PL implementation increases from 10.7% to 72.8%. With the NO\_WAIT policy [42] to prevent deadlocks, the abort rate grows quickly with the number of tuples accessed. At 2000 tuples per transaction, the abort rate is high, causing the overhead distribution to change greatly because overheads grow differently. Some overheads like writing the log buffer only occur after a transaction finishes, some overheads like tuple tracking occurs linearly in the number of tuples accessed, and some overheads like getting the lock are more sensitive to the growing contention. At 2,000 tuples per transaction, the LV updating overhead is around 2.1%.

**Number of Log Files** We also evaluate the effectiveness of the SIMD optimizations. We run Taurus command logging with SIMD on and off against the YCSB-10G workload with 64 threads. The results are shown in Fig. 17. The x-axis is the number of log files Taurus used, and the y-axis represents the time (in nanoseconds) consumed in the LV operations and updates per transaction. The gap between the two baselines increases with the number of log files. Turning on SIMD reduces the overhead by up to 89.5%.

## 6 Related Work

**Early-lock-release (ELR):** ELR [8, 13, 23, 32] allows a transaction to release locks before flushing to log files. Controlled Lock Violation [12] is similar. It permits the acquisition of locks if the lock holder has appended its log record into the buffer. Taurus includes ELR in its design for high performance.

**Single-Storage Logging Algorithms:** ARIES [29] has been the gold standard in database logging and has been widely implemented. However, ARIES does not scale well on multicore processors, as many recent works have observed [16, 35, 37, 43]. C-ARIES [33]

was proposed to support parallel recovery, and CTR [2] improves the recovery time by using multi-versioned concurrency control and aggressive checkpointing, but the contention caused by the original ARIES logging remains.

Aether [16], ELED A [18], and BORDER-COLLIE [20] have optimized ARIES by reducing the length of critical sections during logging. But these protocols still use a single storage device and suffer from the centralized LSN bottleneck. TwinBuf [28] uses two log buffers to support parallel buffer filling. Besides the single storage bottleneck, TwinBuf relies on global timestamps to order the log records. Aether, ELED A, BORDER-COLLIE, and TwinBuf are similar to the serial data baseline we evaluated in Sec. 5.

**Single-Stream Parallel Logging Algorithms:** P-WAL [30] realizes parallel logging but relies on a single LSN counter to order transactions, which will incur scalability issues. Besides, the enforced order causes serial recovery. Adaptive logging [40] achieves parallel recovery for command logging in a distributed partitioned database. Different from Taurus, it infers dependency information from the transactions’ read/write set. This approach requires that the DBMS maintain each transaction’s start and end times to detect dependencies. PACMAN [38] enables parallel command logging recovery by using program analysis techniques to determine what computation can be performed in parallel, while Taurus does not require any program analysis and is simpler to implement. Also, Taurus supports both parallel logging and recovery, while [38] only supports parallel recovery.

**Logging Algorithms for Modern Hardware:** Fast recovery based on non-volatile memory (NVM) is an active research area [3, 4, 6, 10, 15, 21, 22, 36]. This line of work leverages the high read/write bandwidth and byte-addressable nature of NVM to improve the logging and recovery performance. Taurus, in contrast, can be applied to both traditional HDD/SSD devices and the new NVM devices. Since NVM devices are randomly accessible, Taurus can work with multiple log files per NVM device.

**Dependency-Tracking Algorithms:** Similar to Taurus, [8] also uses dependency tracking to log to multiple log files, but does not log dependency information to the log records. This leads to two shortcomings: (1) transactions with dependencies have to be logged in order, which leads to significant performance overhead when there are many inter-log dependencies; (2) they do not support parallel recovery. DistDGCC [41] is also coupled with a dependency tracking logging scheme, but it logs fine-grained dependency graphs. A transaction visiting lots of items results in a huge log record, which may potentially incur scalability issues. In contrast, Taurus only logs LSN Vectors to enforce dependencies. In [17], Johnson et al. proposed a parallel logging scheme that relies on a single-dimension Lamport clock to achieve a global total order of transactions. Taurus uses multi-dimension vector clocks and only preserves partial orders between dependent transactions, enabling moderate parallelism in recovery. Enforcing a total order can accelerate the recovery if the inherent parallelism is significantly low, where a serial recovery is expected to outperform parallel recovery because of the extra inter-thread communications. Taurus provides a serial recovery fallback to fit low-parallelism workloads.

Kuafu [14] presents an algorithm for replaying transactions in parallel on a secondary database. Similar to Taurus, Kuafu also encodes dependency information in the log to replay transactions in parallel. Kuafu supports data logging but not command logging, and it maintains the whole dependency graph while Taurus maintains only minimal dependency information. Therefore, Kuafu will suffer from a bandwidth bottleneck if applied to our setting.

**Multi-Partition Logging:** Bernstein et al. present a logging algorithm [5] for multi-partition databases. They distinguish transactions that only visit a single partition from those that visit multiple partitions. These single-partition transactions are sent to the log file corresponding to the partition (called a single-partition log) as no cross-partition dependencies will occur. All the transactions that visit multiple partitions are sent to a single log file (called the multi-partition log). Their design also uses vector clocks—two-dimensional vector clocks are maintained by each log to keep a partial order between the multi-partition log and the corresponding single-partition log. Given a partitioning scheme, picking out transactions without cross-log dependencies is orthogonal to the problem we solve here; Taurus can be plugged in to better deal with multi-partition transactions by enabling multiple-stream logging.

## 7 Conclusion

We presented Taurus, a lightweight parallel logging scheme for high-throughput main memory DBMSs. It is designed to support not only data logging but also command logging, and is compatible with multiple concurrency control algorithms. It is both efficient and scalable compared to state-of-the-art logging algorithms.

## References

- [1] DBx1000. <https://github.com/yxymit/DBx1000>.
- [2] P. Antonopoulos, P. Byrne, W. Chen, C. Diaconu, R. T. Kodandaramaihi, H. Kodavalla, P. Purnananda, A.-L. Radu, C. S. Ravella, and G. M. Venkataramanappa. Constant time recovery in azure sql database. *Proceedings of the VLDB Endowment*, 12(12):2143–2154, 2019.
- [3] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let’s talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, pages 707–722, 2015.
- [4] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *VLDB*, 10(4):337–348, 2016.
- [5] P. A. Bernstein and S. Das. Scaling optimistic concurrency control by approximately partitioning the certifier and log. *IEEE Data Eng. Bull.*, 38(1):32–49, 2015.
- [6] A. Chatzistergiou, M. Cintra, and S. D. Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *VLDB*, 8(5):497–508, 2015.
- [7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, pages 143–154, 2010.
- [8] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. *Implementation techniques for main memory database systems*, volume 14. ACM, 1984.
- [9] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server’s Memory-Optimized OLTP Engine. In *SIGMOD*, pages 1243–1254, 2013.
- [10] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1221–1231. IEEE, 2011.
- [11] C. J. Fidge. *Timestamps in message-passing systems that preserve the partial ordering*. Australian National University. Department of Computer Science, 1987.
- [12] G. Graefe and H. Kuno. Controlled lock violation for data transactions, July 19 2016. US Patent 9,396,227.
- [13] G. Graefe, M. Lillibridge, H. Kuno, J. Tucek, and A. Veitch. Controlled lock violation. In *SIGMOD*, pages 85–96. ACM, 2013.
- [14] C. Hong, D. Zhou, M. Yang, C. Kuo, L. Zhang, and L. Zhou. Kuafu: Closing the parallelism gap in database replication. In *ICDE*, pages 1186–1195. IEEE, 2013.
- [15] J. Huang, K. Schwan, and M. K. Qureshi. Nvram-aware logging in transaction systems. *VLDB*, 8(4):389–400, 2014.
- [16] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *VLDB*, 3(1-2):681–692, 2010.
- [17] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Scalability of write-ahead logging on multicore and multisoocket hardware. *The VLDB Journal*, 21(2):239–263, 2012.
- [18] H. Jung, H. Han, and S. Kang. Scalable database logging for multicores. *VLDB*, 11(2):135–148, 2017.
- [19] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.
- [20] J. Kim, H. Jang, S. Son, H. Han, S. Kang, and H. Jung. Border-collie: A wait-free, read-optimal algorithm for database logging on multicore hardware. In *SIGMOD*, pages 723–740. ACM, 2019.
- [21] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. Nvwal: Exploiting nvram in write-ahead logging. *ACM SIGOPS Operating Systems Review*, 50(2):385–398, 2016.
- [22] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *SIGMOD*, pages 691–706. ACM, 2015.
- [23] H. Kimura, G. Graefe, and H. A. Kuno. Efficient locking techniques for databases on modern hardware. In *ADMS@ VLDB*, pages 1–12, 2012.
- [24] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.
- [25] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *VLDB*, pages 298–309, 2011.
- [26] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *ICDE*, pages 604–615, March 2014.
- [27] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226, 1988.
- [28] Q. Meng, X. Zhou, S. Wang, H. Huang, and X. Liu. A twin-buffer scheme for high-throughput logging. In *International Conference on Database Systems for Advanced Applications*, pages 725–737, 2018.
- [29] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.
- [30] Y. Nakamura, H. Kawashima, and O. Tatebe. Integration of tictoc concurrency control protocol with parallel write ahead logging protocol. *International Journal of Networking and Computing*, 9(2):339–353, 2019.
- [31] D. A. Patterson, G. Gibson, and R. H. Katz. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, volume 17. ACM, 1988.
- [32] E. Soisalon-Soininen and T. Ylönen. Partial strictness in two-phase locking. In *International Conference on Database Theory*, pages 139–147. Springer, 1995.
- [33] J. Speer and M. Kirchberg. C-aries: A multi-threaded version of the aries recovery algorithm. In *International Conference on Database and Expert Systems Applications*, pages 319–328. Springer, 2007.
- [34] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0), June 2007.
- [35] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-Memory Databases. In *SOSP*, 2013.
- [36] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *VLDB*, 7(10):865–876, 2014.
- [37] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *EuroSys*, page 26, 2014.
- [38] Y. Wu, W. Guo, C.-Y. Chan, and K.-L. Tan. Fast Failure Recovery for Main-Memory DBMSs on Multicores. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 267–281, 2017.
- [39] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 169–182, 2020.
- [40] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *SIGMOD*, pages 1119–1134, 2016.
- [41] C. Yao, M. Zhang, Q. Lin, B. C. Ooi, and J. Xu. Scaling distributed transaction processing and recovery based on dependency logging. *VLDB Journal*, 27(3):347–368, 2018.
- [42] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. pages 209–220, 2014.
- [43] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*, pages 465–477, 2014.
- [44] H. Zhou, J. Guo, H. Hu, W. Qian, X. Zhou, and A. Zhou. Plover: parallel logging for replication systems. *Frontiers of Computer Science*, 14(4):144606, 2020.

## A Proof of Correctness & Liveness

In this section, we prove both the correctness and liveness of the Taurus protocol. Specifically, we prove that (1) data dependencies are correctly enforced during recovery; (2) all and only committed transactions will be recovered; and (3) the protocol will not deadlock or livelock during forward execution and recovery if the concurrency control protocol does not deadlock or livelock. We use  $T.LSN$  to denote  $T$ 's position in the corresponding log.

**THEOREM 1. [Correctness of Recovery Order]** *Data dependencies are correctly enforced during recovery: for any transaction  $T_2$  that depends on  $T_1$ ,  $T_2$  is recovered after  $T_1$ .*

**PROOF.** We prove the theorem in two steps. First, we prove that for any transaction  $T_2$  that depends on  $T_1$ , Taurus enforces that  $T_2.LV[i] \geq T_1.LSN$ , where  $T_1$  logs to the  $i$ -th log manager (i.e.,  $L_i$ ). Then, we prove that if  $T_2.LV[i] \geq T_1.LSN$ ,  $T_2$  will be recovered after  $T_1$ .

**Step 1:** W.l.o.g., we consider a RAW dependency where  $T_1$  writes to tuple  $A$  and then  $T_2$  reads  $A$  (proofs for write-after-read or write-after-write dependencies are similar). According to Lines 10, 11 and 16 in Alg. 1,  $A.writeLV[i] = T_1.LSN$  when  $T_1$  releases its write lock on  $A$ . When  $T_2$  reads  $A$  at a later time, Line 3 in Alg. 1 enforces that  $T_2.LV[i] \geq A.writeLV[i]$ . Since  $writeLV$  can only monotonically increase, we have  $T_2.LV[i] \geq T_1.LSN$ .

**Step 2:** During recovery,  $T_2$  can be recovered only if  $T_2.LV \leq RLV$ , which means  $T_2.LV[i] \leq RLV[i]$  (Line 2 in Alg. 4). Given  $T_1.LSN \leq T_2.LV[i]$ , the recovery of  $T_2$  requires  $T_1.LSN \leq RLV[i]$ . According to Lines 4–7 in Alg. 4, all transactions in  $L_i$  with LSNs no greater than  $RLV[i]$  have been recovered. Therefore,  $T_1.LSN \leq RLV[i]$  means  $T_1$  is already recovered. The recovery of  $T_2$  requires the recovery of  $T_1$ .  $\square$

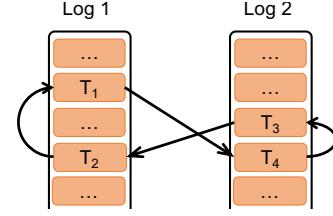
**THEOREM 2. [Correctness of Completeness]** *All and only committed transactions will be recovered.*

**PROOF.** Given that Taurus *in-order* commits transactions that map to the same log manager (Line 18 in Alg. 1), we must find the last committed transaction  $T$  and prove that  $T$  and all transactions before  $T$  are recovered and no transaction after  $T$  is recovered.

Based on Line 1 in Alg. 3, the transaction  $T$  that we are looking for is the transaction right before the first transaction  $T'$  that violates  $T'.LV \leq ELV$ . If no such  $T'$  exists, we choose  $T$  as the last transaction in the log file. Given this, we prove the following three properties:

**Property #1:**  $T$  is the last committed transaction before the crash. Since  $T$  is the transaction right before the first transaction  $T'$  violating  $T'.LV \leq ELV$ , all transactions before  $T'$  satisfy this inequality. Therefore, before the crash, for all of them we have  $T.LV \leq PLV$ . By Line 18 in Alg. 1, these transactions have committed before the crash. We then prove that  $T'$  did not commit before the crash. There are two cases to consider. If  $T'$  does not exist in the log file, then  $T'$  was not persistent and thus never committed before the crash. Otherwise,  $T'$  exists but  $T'.LV > ELV$ . This means during forward processing, we have  $T'.LV > PLV$ , indicating that  $T'$  never committed before the crash.

**Property #2:** All transactions before  $T'$  will be recovered. During recovery,  $T$  and transactions before  $T$  form a directed dependency graph where each element in an LV indicates an edge in the



**Figure 18: A Hypothetical Deadlock Situation** – The dependencies of transactions form a cycle.

graph. Later, in Theorem 4, we prove that the dependency graph is acyclic, which means that each transaction will be recovered.

**Property #3:** No transaction after  $T$  will be recovered. By Line 1 in Alg. 3, all transactions after  $T$  are ignored and will not be recovered.  $\square$

**THEOREM 3. [Liveness in Forward Processing]** *The protocol will not deadlock or livelock during forward processing if the concurrency control protocol does not deadlock or livelock.*

**PROOF.** Assuming that no thread will be indefinitely suspended, we show that the system will eventually make progress (e.g., committing a transaction). We prove this in three parts:

First, if there are transactions in the log buffer, they will eventually be flushed to disks. The only possibility that data in the log buffer is not flushed is because  $readyLSN$  is throttled by  $allocatedLSN$  (Line 4 in Alg. 2). This throttling can happen only if  $allocatedLSN[j] \geq filledLSN[j]$  which means a transaction is in the middle of filling a log record. This process, however, completes in a short period of time, since all the operations between Lines 20 and 23 in Alg. 1 are non-blocking.

Second, active transactions will eventually be written to the log buffer. Assuming the concurrency control algorithm does not incur deadlocks/livelocks, the logic in Alg. 1 is combinational and non-blocking, and the conditional statements are independent from the concurrency control algorithm. Since a log buffer will finally flush, it will contain space for an active transaction to write to.

Finally, an active transaction will eventually commit assuming no system failure. Since transactions in each log buffer will eventually be flushed, for each transaction  $T$ ,  $PLV$  will exceed  $T.LV$ . This will also be true for transactions in the same log manager with smaller LSNs. Following Line 18 in Alg. 1, this means  $T$  will eventually commit.  $\square$

**THEOREM 4. [Liveness in Recovery]** *The protocol will not deadlock or livelock during the recovery process.*

**PROOF.** The recovery follows a directed dependency graph where each node is a transaction and each edge corresponds to a value in  $LV$ . Now, we prove that the graph is acyclic.

Fig. 18 shows a hypothetical deadlock situation that may occur if Taurus is incorrectly designed. A cycle is formed between the four transactions:  $T_1 \leftarrow T_2 \leftarrow T_3 \leftarrow T_4 \leftarrow T_1$ . Although all the transactions are in the *pools*, none of them is able to make any forward progress. Note that, the correctness of the concurrency control algorithm is not sufficient to rule out this situation because Taurus adds extra dependencies while logging.

To prove that no dependency cycles exist in Taurus, we define a commit time,  $ct(T)$ , to each transaction  $T$ . We show that every edge



in the graph follows the order of commit time, namely, an edge  $T2 \rightarrow T1$  exists  $\Rightarrow ct(T2) > ct(T1)$ . Since time specifies a total order, proving this inequality means cycles are impossible. In particular, we choose the time when  $T$ 's log record is allocated in the log buffer (i.e., right after atomically incrementing LSN in Alg. 1, Line 21) as its commit time  $ct(T)$ .

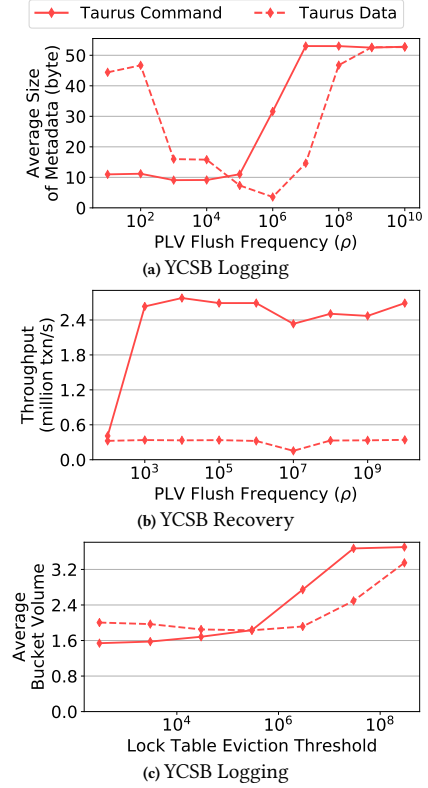
As we proved in Theorem 1,  $T2$  depending on  $T1$  mapped to log manager  $i \Rightarrow T2.LV[i] \geq T1.LSN$ . It is clear that for transactions mapping to the same log manager, their  $ct$  order is the same as their LSN order; therefore  $ct(T') > ct(T1)$ , where  $T'$  is the transaction on  $L_i$  with  $T'.LSN = T2.LV[i]$ . According to Line 11 in Alg. 1, a tuple may have its  $readLV[i]$  or  $writeLV[i]$  equal a particular LSN only after the log record has been written to the log buffer at that LSN.  $T2.LV[i]$  must be copied from a tuple and thus must occur at a even later time, so  $ct(T2) > ct(T') > ct(T1)$ . Therefore, if  $T2.LV[i] \geq T1.LSN$ , we have  $ct(T2) > ct(T1)$  proving the theorem.  $\square$

## B Correctness Proof of LV Compression

We next prove that the theorems in Appendix A still hold with the two optimizations discussed above. Both optimizations share the same basic idea: if a  $LV$  is too small, it suffices to store an upper bound of it, which can be shared by multiple  $LV$ s. Therefore, for a transaction  $T$  with  $LV$  before the optimizations and  $LV'$  after the optimizations, we must have  $LV' \geq LV$ . The optimizations will not affect the correctness of Theorem 1 since it does not violate existing dependencies. Theorem 2 is also not affected since  $LV'$  will never exceed  $PLV$  and thus  $ELV$  as well. Theorem 3 is not affected by the second optimization; for the first optimization, it will not block the asynchronous commit of transactions since an increased  $LV$  of a transaction is not higher than the current  $PLV$ . Finally, Theorem 4 is not as straightforward – since the optimizations make transactions depend on more transactions, it may potentially create cycles in the dependency graph. In the following, we will prove that although it increases  $LV$ s, it only increases them to the point that the dependencies still follow the real-time order. Therefore, following the proof of Theorem 4, no cycle may exist.

For the first optimization, the DBMS may copy the tuple's  $LV$  from  $PLV$ . For the second optimization, the DBMS increases some dimensions of a log record's  $LV$  to the corresponding values of the  $PLV$ . In both cases, the log record of a transaction  $T$  will have an  $LV$  no greater than the current  $PLV$  when the DBMS writes  $T$ 's log records to the buffer. Therefore, for each  $T'$  in  $Log\ i\ s.t.\ T'.LSN \leq T.LV[i]$ , we still have  $ct(T) > ct(T')$ .

As an intuition, the increment of a tuple's  $LV$  is equivalent to having a dummy transaction that reads a tuple and later writes the same values back to it. Similarly, the increment of a transaction's  $LV$  is equivalent to having the transaction visit a dummy tuple with  $PLV$  as its  $LV$ . Inserting such dummy transactions or visiting dummy tuples only causes  $LV$ s to artificially increase in the same way. These operations do not affect the correctness of the database, because the same tuple accesses and transaction interleavings form valid runtime events for the un-optimized Taurus described in Sec. 3, and it can handle them by the proofs in Appendix A.



**Figure 19: LV Compression** – (a) PLV Flush Frequency ( $\rho$ ) vs. the average size of metadata in a high-contention workload; (b) PLV Flush Frequency ( $\rho$ ) vs. the recovery throughput; and (c) lock table eviction threshold vs. the average bucket volume.

## C Evaluation of the LV Compression

We next evaluate the scalability optimizations that we presented in Sec. 4.1 with a high-contention workload. We use a single byte to denote the number of elements in the *compressedLV*, and a 64-bit integer for each element. Without compression, the  $LV$  would take 64 bytes to store the eight 64-bit integers. We test the effect of the compression with long YCSB transactions (each visiting 16 rows). We measure how the amount of metadata is affected when adjusting the flush frequency. We vary the *PLV Flush Frequency*  $\rho$  through a large range. The DBMS writes a  $PLV$  anchor for every  $\rho$  bytes of the log. The DBMS tracks both the LSN Vectors of transactions and the  $PLV$ s when computing the metadata size.

From Fig. 19a, we observe that even for a high-contention workload, when we set  $\rho$  appropriately, on average the DBMS only has to write 3.5 bytes of metadata per log record for Taurus data logging, and 9.1 bytes per log record for Taurus command logging. When  $\rho$  is small, the average metadata size can be significant because the DBMS flushes too many  $PLV$ s, causing extra overhead amortized on each record. As *PLV Flush Frequency* grows beyond  $10^6$ , the size of the metadata increases and finally reaches a steady value. The Taurus data logging curve is “right-shifted” compared to that of Taurus command logging because a data logging record is about 26 $\times$  larger than a command logging record. Therefore, the same  $\rho$  results in more frequent  $PLV$  flushes for command logging than data logging. We see in Fig. 19b that a larger  $\rho$  tends to bring

better recovery performance<sup>1</sup> because the *PLVs* are flushed less frequently, resulting in fewer artificial dependencies. This is not observed in Taurus data logging as the recovery is bounded by the I/O bandwidth.

We also examine how the DBMS's lock table eviction threshold  $\delta$  affects the average volume of the lock table buckets. The results in Fig. 19c indicate that a larger  $\delta$  threshold results in a

larger lock table. Qualitatively, when  $\delta$  is large, we expect Taurus to perform better during the recovery because the DBMS enforces fewer extra dependencies. We contend that such a difference is only possible when the recovery manager is not the bottleneck, and the underlying workload has a moderate level of contention.

<sup>1</sup>To fully exploit the recovery parallelism, we use more threads in recovery than in logging: 56 workers in recovery and 8 workers in logging.