Taurus: Lightweight Parallel Logging for In-Memory Database Management Systems

Yu Xia, Xiangyao Yu, Andrew Pavlo, Srinivas Devadas
Massachusetts Institute of Technology, University of Wisconsin–Madison, Carnegie Mellon University
yuxia@mit.edu,yxy@cs.wisc.edu,pavlo@cs.cmu.edu,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.

PVLDB Reference Format:
Yu Xia, Xiangyao Yu, Andrew Pavlo, Srinivas Devadas. Taurus: Lightweight Parallel Logging for In-Memory Database Management Systems. PVLDB, 14(2): 189-201, 2021.
doi:10.14778/3425879.3425889

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, 44] 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.

Proceedings of the VLDB Endowment, Vol. 14, No. 2 ISSN 2150-8097.
doi:10.14778/3425879.3425889

There are several parallel logging and recovery proposals in the literature [16, 35, 37, 44]. 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, 44]; 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 serial logging scheme (with and without RAID-0 setups) and state-of-the-art parallel logging schemes (i.e., Silo-R [35, 44] and Plover [45]) on YCSB and TPC-C benchmarks. Our evaluation on eight NVMe SSDs shows that Taurus with data logging outperforms serial data logging by 9.9× at runtime, and Taurus with command logging outperforms the serial command logging by 2.9×. During recovery, Taurus with data logging and command logging is 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 is 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 the extended version [39].
- 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 parallel logging schemes, and demonstrate its advantages and generality.
- We open source Taurus and evaluation scripts at https://github.com/yuxiamit/DBx1000_logging.

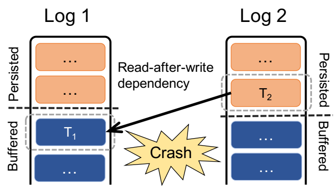


Figure 1: Data Dependency in Parallel Logging — Transaction *T2* depends on *T1*. The two transactions write to different logs.

2 Background

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

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 *T2* depends on *T1*, then the DBMS writes *T2* to disk after *T1*. A transaction commits only after it successfully writes the transaction's log records to disk. During recovery, the DBMS reads the log and replays each transaction sequentially until it encounters an incomplete log record or the end of the file.

Generally, 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 reapplies these changes to the database. The other category, *command logging* [26], reduces the amount of log data by only recording the high-level commands (i.e., invocations of stored procedures). The log records for these commands are typically smaller than physical changes. The recovery process involves more computation, as all transactions are re-executed. If 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, 44]. 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 scalability bottleneck [35]. Competing for the single atomic LSN counter inhibits performance due to cache coherence traffic [43].

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 *T1* and *T2*, where *T2* depends on *T1* with a read-after-write (RAW) data dependency. In this example, we assume that *T1* writes to *Log 1* and *T2* writes to *Log 2* and they may be flushed in any order. If *T2* is already persistent in *Log 2* while *T1* is still in the log buffer (shown in Fig. 1), the DBMS must *not* commit *T2* since *T1* has not committed. Furthermore, if the DBMS crashes, the recovery process must be aware of such data dependency and therefore should not recover *T2*. 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 on other log streams.

Challenge #2 – Whether to Recover a Transaction: *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. It ignores any transaction that fails to complete properly. For the example in Fig. 1, if *T2* is in the log but *T1* is not, the DBMS should not recover *T2*.

Challenge #3 – Determine the Recovery Order: The DBMS must recover transactions in the order that respects data dependencies. If both *T1* and *T2* in Fig. 1 are persistent and have committed before the crash, the DBMS must recover *T1* before *T2*.

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, 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, it only 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 *T1* that writes A=1, and a transaction *T2* that reads A and then writes B=A+1. Suppose the initial value of A is 0, and the DBMS schedules *T2* before *T1*, resulting in A=1 and B=1. With this schedule, *T1* has a WAR dependency on *T2*. If the DBMS does not track WAR dependencies and perform command logging, running *T1* before *T2* will result in A=1 and B=2, violating correctness. But if the DBMS performs data logging, *T1* will have a record of A=1 and *T2* will have a record of B=1. Regardless of the recovery order between *T1* and *T2*, the resulting state is always correct. Supporting only data logging simplifies the protocol [35, 44]. 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.

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. 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. We also assume that the DBMS uses multiple disks with

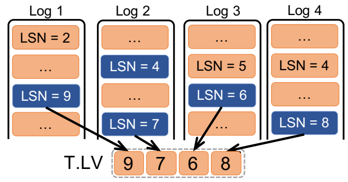


Figure 2: LSN Vector (LV) example — The i^{th} element of transaction T's LV is an LSN of the i-th log, indicating that T depends on one or more transactions (rendered in dark blue) in the i-th log before that LSN.

each log file residing on one disk. Each transaction writes only a single log entry to one log file at commit time. This simplifies the protocol and is used by other in-memory DBMSs [9, 19, 35, 44].

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 track 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: $T.LV = (LV[1], LV[2], \ldots, LV[n])$, 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 with $T.LV$ [2]=7. It means that T may depend on any transaction that maps to *Log 2* with an LSN ≤ 7 but no transaction with an LSN > 7. The semantics of LV is similar to vector clocks [11, 27]. The following two operations will be frequently used on LVs: *ElemWiseMax* and *comparison*. The *ElemWiseMax* is the element-wise maximum function:

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

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 LVs is sufficient to resolve the challenges in Sec. 2.2: (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 of $T.LV$. (3) The recovery order follows the partial order specified by LVs.

3.2 Logging Operations

The Taurus protocol runs on *worker* threads and *log manager* threads (denoted as $L_1, L_2, \ldots, L_n$). 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. We first describe the protocol's 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 $T.LV$ tracking its dependency as in Sec. 3.1. Initially, $T.LV$ 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 Taurus maintains them in the lock table (cf. Sec. 4.1).
- *L.logLSN* – The highest position that has not been allocated in the log file of 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 L. Initially, all elements are ∞.
- *L.filledLSN* – A vector of length p, storing the last LSN filled by each worker of 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.
- *Global.PLV* – *PLV* stands for *Persistent LSN Vector*. It is a global vector of length n. The element PLV_i 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., no-wait). After a worker thread acquires a lock, it runs Lines 3–5 to update the LV of the transaction: It first updates $T.LV$ to be no less than the LV of previous writing transactions to reflect WAW and RAW dependencies. If the access is a write, it also updates $T.LV$ using the tuple's *readLV*.

The DBMS calls the *Commit* function when the transaction finishes. At this moment, T has locked the tuples it accessed. Since Taurus updates $T.LV$ for each access, it already captures T's dependency information. It checks if T is read-only, and skips generating log records if so. Otherwise, it creates the log record for T (Line 8). The record contains two parts: the *redo log* and a copy of *T's current LV*. The contents of the redo log depends on the logging scheme: the keys and values (for data logging), or the high-level command (for command logging). The DBMS writes the record into the corresponding log manager's buffer by *WriteLogBuffer* (Line 10). The algorithm then updates $T.LV[i]$ 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 stays the same. Lines 13–17 update the metadata of the tuples before releasing the locks. If T reads (writes) a tuple, it updates the tuple's *readLV* (*writeLV*) using $T.LV$, indicating that the tuple was read (written) by T and future transactions must respect this dependency. Updating the LVs and releasing the lock must be executed atomically, otherwise multiple transactions concurrently updating the *readLV* can cause race conditions leading to incorrect dependencies. As most 2PL schemes use latches to protect lock release, updating LVs can be piggybacked within those latches.

Algorithm 1: Worker Thread with index j for log 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
           # Include T's LV into the log record.
8          logRecord = {CreateLogRecord(T), copy(T.LV)};
9          recordSize = GetSize(logRecord);
10         LSN = WriteLogBuffer(logRecord, recordSize);
11         T.LV[i] = LSN# Update T.LV[i] in the memory.;
12     for key ∈ T's access set do
13         if T reads DB[key] then                    # Atomic Section
14             DB[key].readLV = ElemWiseMax(T.LV, DB[key].readLV);
15         if T writes DB[key] then
               # T.LV is always no less than DB[key].writeLV
16             DB[key].writeLV = T.LV;
17         Release(key)
18     Asynchronously commit T if PLV ≥ T.LV and all transactions in
         L_i with smaller LSNs have committed;

19 Function WriteLogBuffer(logRecord, recordSize)
20     L_i.allocatedLSN[j] = L_i.logLSN;
21     lsn = AtomicFetchAndAdd(L_i.logLSN, recordSize);
22     memcpy(L_i.logBuffer + lsn, logRecord, recordSize);
23     L_i.filledlSN[j] = lsn + recordSize;
24     return lsn + recordSize
```

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 an entry into the log buffer. It 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 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 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.

To show how Taurus tracks dependencies, we use the example in Fig. 3 with three transactions (*T1*, *T2*, *T3*) and two rows A,B. WLOG, we assume *T1* and *T2* are assigned to Log 1 and *T3* 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

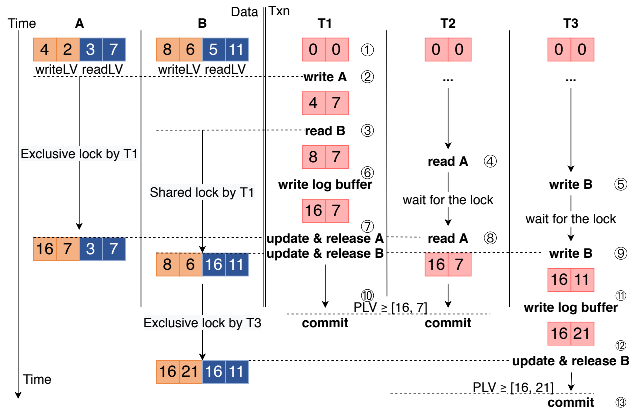


Figure 3: Worker Thread Example. Three transactions (*T1*, *T2*, and *T3*) are accessing two objects A and B. Transactions are logged to two files. The diagram is drawn in the time order.

transactions' *LV*s as [0,0]. ② *T1* acquires an exclusive lock on A and writes to it. Then, *T1* updates *T1.LV* to be the element-wise maximum among A.*writeLV*, A.*readLV*, and *T1.LV*. In this example, *T1.LV*=[max(4,3,0), max(2,7,0)] = [4,7]. Any previous transactions that ever read or wrote A should have an *LV* no greater than *T1.LV*. ③ *T1* acquires a shared lock on B and then reads it. Then, *T1* updates *T1.LV* to be the element-wise maximum among *B.writeLV* and *T1.LV*. Now *T1.LV*= [8,7]. ④ *T2* wants to read A but has to wait for *T1* to release the lock. ⑤ *T3* wants to write B but has to wait as well. ⑥ After *T1* finishes, *T1* writes its redo record and a copy of *T1.LV* into the log buffer. After successfully writing to the buffer, *T1* learns its LSN in Log 1 is 16. Then, *T1* updates the first dimension of *T1.LV* to be 16. Now, *T1.LV*=[16,7]. ⑦ *T1* updates *A.writeLV* = *ElemWiseMax*(*A.writeLV*, *T1.LV*) = *T1.LV* = [16,7], and *B.readLV* = *ElemWiseMax*(*B.readLV*, *T1.LV*) = [16,11]. Then, *T1* releases the locks. After this, *T1* waits for itself and all the transactions it depends on to become persistent, equivalently, *PLV* $\geq$ *T1.LV*. The thread can process other transactions, and periodically check if *T1* should be marked as committed. ⑧ *T2* acquires the shared lock on A. *T2* then updates *T2.LV*=*ElemWiseMax* (*T2.LV*, A.*writeLV*) = [16,7]. This update enforces the partial order that *T1.LV* $\leq$ *T2.LV* because *T2* depends on *T1*. Since *T2* is read-only, it does not create a log record. It enters the asynchronous commit by waiting for *PLV* $\geq$ *T2.LV*. ⑨ *T3* acquires an exclusive lock on B and updates *T3.LV* = *ElemWiseMax*(*T3.LV*, *B.readLV*, *B.writeLV*) = [16,11]. The fact that *T3* depends on *T1* reflects on *T3.LV* $\geq$ *T1.LV*. ⑩ The logging threads have flushed all transactions before *T1.LV* = *T2.LV* = [16,7] and updated *PLV*. Observing *PLV* $\geq$ [16,7], Taurus marks *T1* and *T2* as committed. ⑪ *T3* writes its redo record and a copy of *T3.LV* to the buffer of Log 2, and gets its LSN as 21. *T3.LV* increases to [16,21]. ⑫ *T3* sets B.*writeLV* to [16, 21] and releases the lock. ⑬ When *PLV* achieves *T3.LV* = [16, 21], Taurus commits *T3*.

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. The algorithm identifies up to which point of the buffer that no active worker threads are processing.

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] ≥ filledLSN[j] then
4         readyLSN = min(readyLSN, allocatedLSN[j])
5 flush the buffer up to readyLSN;
6 PLV[i] = readyLSN;
```

Taurus uses *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 L_i*.logLSN* (Line 1). For each worker thread j that belongs to L_i, if *allocatedLSN*$[j] \geq$ *filledLSN*$[j]$, the transaction in thread j is filling the 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 up to *readyLSN* and updates *PLV*$[i]$.

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 ambiguity of transaction dependency during recovery. Sequential committing will not affect scalability because ELR prevents transactions waiting on the critical path.

3.3 Recovery Operations

Taurus' recovery algorithm replays transactions following the partial orders between their *LV*s, sufficient to respect all the data dependencies. Resolving the recovery order is equivalent to performing topological sorting in parallel on a dependency graph.

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 (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). A transaction T committed before the crash if $T.LV \leq ELV$. Otherwise, T and transactions after it are ignored for recovery. The 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 crucial that the update of *maxLSN* occurs

Algorithm 3: Log Manager Recovery for Thread L_i.

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

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)
```

after it runs *Enqueue*, otherwise the transactions may recover 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; a worker might recover a transaction that depends on T before T itself 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 no conflicts will occur during recovery. Then, *RLV*$[i]$ is updated (Lines 4-7). If *pool* is empty, the thread sets *RLV*$[i]$ to *pool.maxLSN*, the largest LSN of any transaction added to *pool*; 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 updates the *readLV*'s of the rows. To avoid phantoms, the transaction performs the same scan again before releasing the locks in *Commit*. If the result rows are different, 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. In the second scan, the rows in the previous scan still exist because of the shared locks. Therefore, if the row count remains 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 the *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 $T.LV = ElemWiseMax(T.LV, DB[key].readLV, DB[key].writeLV)$. 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

One potential issue is that the size of LV is linear 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 use a relatively small number of log files and thus this overhead is acceptable, Taurus can leverage LV compression (Sec. 4.1) and SIMD (Sec. 5.6) to mitigate this issue.

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

Lastly, to exploit parallelism in recovery, workers might need to scan the whole pool to find the next transaction ready to recover. Heuristic optimizations like zig-zag scans could help. We defer developing a data structure for Taurus recovery to future work.

4 Optimizations and Extensions

We now discuss optimizations to reduce overhead, and 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, potentially increasing the log size especially for command logging where each entry is relatively small. We describe optimizations that address these problems.

Tuple LV Compression: Keeping LVs for tuples accessed a long time ago is unnecessary. Their LVs are too small to affect active transactions. This optimization thus stores LVs only for active tuples in the lock table. Transactions operate on their LVs following Alg. 1. If the DBMS inserts a tuple into the lock table, it assigns its *readLV* and *writeLV* to be the current *PLV*. The system can evict a tuple from the lock table if no transactions hold locks on it and both its *readLV* and *writeLV* are not greater than *PLV*.

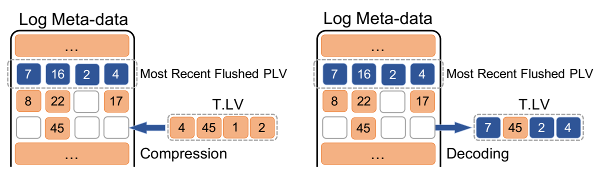
For the tuples previously evicted from the lock table and later inserted back, the optimization increases the *readLV* and *writeLV* of these tuples and also the LVs of transactions accessing them. This causes unnecessary dependencies. To make the trade-off between higher compression ratio and fewer artificial dependencies, we introduce a parameter δ 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 δ means fewer artificial dependencies, but more tuples will stay in the lock table, and vice versa.

Log Record LV Compression: We next reduce the log storage. We let each log record store only a part of the 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. Therefore, we can set anchor points (in the form of LVs) into logs and let T only store the elements in LV beyond the latest anchor point.

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.

Algorithm 5: *LV* Compression for Log Records

```
Function FlushPLV()
    currentPLV = Global.PLV;
    logBuffer.append(currentPLV);
    LPLV = currentPLV;
Function Compress(LV)
    compressedLV = LV;
    foreach LV[j] ∈ LV do
        if LV[j] ≤ L_i.LPLV[j] then
            compressedLV[j] = NaN;
    return compressedLV;
Function Decompress(compressedLV)
    LV = compressedLV;
    foreach LV[j] ∈ LV do
        if LV[j] = NaN then
            LV[j] = L_i.LPLV[j];
    return LV;
```



(a) Logging (b) Recovery

Figure 4: LV Compression. Example of Taurus's LV compression.

It keeps a copy of the most recent *PLV* written into L's log buffer. Periodically, the log manager calls *FlushPLV* to append *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 $T.LV[j] \leq L_i.LPLV[j]$, we can increase $T.LV[j]$ to $L_i.LPLV[j]$. Since $L_i.LPLV$ is already in the buffer, the DBMS 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^{st}, 3^{rd}, and 4^{th} 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, more dimensions of *LV* are smaller than *LPLV* and thus it enables better compression, but some amount of recovery parallelism is sacrificed since the decompressed LVs have larger values.

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 ∈ sorted(T.writeSet) do
7          DB[key].lock();
8      for key ∈ 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
         Alg. 1;
15     for key ∈ T.writeSet do
16         DB[key].writeLV = ElemWiseMax (DB[key].writeLV, T.LV);
17         DB[key].release();
18     Asynchronously commit T if PLV ≥ T.LV and all transactions in
         L_i with smaller LSNs have committed;
```

4.2 Optimization: Vectorization

The logging overhead mainly consists of: (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 latency due to (1) and (2) will result in extra lock contention; (4) the time cost in persisting the log records to the disk. All these overheads will not block the DBMS from scaling up. Among them, (2) and (3) are shared by essentially all the write-ahead logging algorithms; (4) is moved off the critical path by ELR. Overhead (1) is linear in the number of log files. In our evaluation with 16 log files, it is up to 13.8% of the total execution time if implemented naively. We can exploit the data parallelism in LSN Vectors as the values in a single vector are processed independently. Modern CPUs provide SIMD extensions that allow processing multiple vector elements 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 vectorized operations reduce Taurus' overhead by 89.5%.

4.3 Extension: Support for OCC

Our overview of Taurus so 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]. Alg. 6 shows the protocol. 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 metadata. Common in OCC algorithms, the *ValidateSuccess* function returns true if the values in the *readSet* are not modified by others. 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 RAW dependencies. To see a failure example, consider a transaction *T1* modifying the data after *T2*'s validation but before *T2*'s updates on $readLV$. Then, it is possible that *T1* does not observe the latest $readLV$, and fails to capture the RAW dependency to *T2*. Note that updating $readLV$ before the validation might cause unnecessary dependencies (i.e., LVs larger than necessary) if the transaction aborts later in the validation. Such aborts only affect performance but not correctness. The log managers stays the same as in Alg. 2.

4.4 Extension: Multi-Versioning

We next discuss how Taurus works with MVCC. Concurrency control algorithms based on logical timestamps allow physically late transactions to access early data versions and commit transactions logically early. However, log records are flushed in the physical time order. Solving the decoupled order requires extra design. Thus, we assume the recovery process also uses multi-versions. This relaxes the decoupling by allowing physically late transactions to commit logically early in the recovery. It also frees Taurus from tracking the WAR dependencies because read operations can still fetch the correct historic version even after the tuple has been modified. Therefore, Taurus only needs to track the WAW and RAW dependencies. Different from Sec. 3.1, Taurus for MVCC only adds a single metadata for the data versions, the LSN Vector LV. Our discussion is based on the MVCC scheme [25] used in Hekaton [9].

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.

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 record 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 decides the visible data version 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 no conflicts will occur. Without Taurus, the logical timestamps in log records enforce a total order. Taurus exploits parallelism to recover non-conflicting transactions in parallel.

5 Evaluation

We implemented both the 2PL and OCC variants of Taurus in the DBx1000 in-memory DBMS [1] to evaluate its performance. We evaluate them on three storage types: (1) NVMe SSDs, (2) HDDs, and (3) Persistent Memory (PM) simulated by a RAM disk. The performance profiles of these devices highlight different properties of Taurus. As the mainstream storage, NVMe SSDs provide high I/O bandwidth, enabling insights into 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.

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 parallel data logging scheme partitions log records based on data accesses [45]. It uses per-log sequence numbers to enforce a total order among transactions. Each transaction generates multiple log entries.

Silo-R: Lastly, we also implemented the parallel logging scheme from Silo [35, 44]. 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

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.

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 each table. The access pattern of transactions visiting the rows follows a Zipfian distribution; we set the distribution parameter to 0.6 to simulate moderate contention. Each transaction accesses two tuples and each access has a 50% chance to be a read operation (otherwise a write operation). We will perform sensitivity studies regarding these workload parameters in Sec. 5.6. The size of a command log record is smaller than that of a data log record.

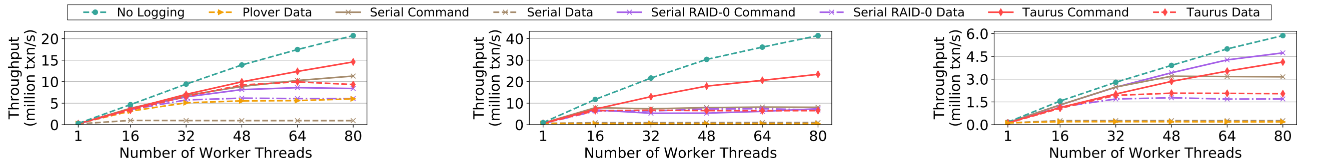
TPC-C: This is the standard OLTP benchmark that simulates a wholesale company operating on warehouses [34]. There are nine tables covering a variety of necessary information and transactions are performing daily order-processing business. We simulate two (Payment and New-Order) out of the five transaction 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 80 warehouses. We evaluate the full TPC-C workload in Sec. 5.5.

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 16 GB/s bandwidth. We use at most 80 worker threads and 16 log manager threads to avoid context switches. Every disk contains two log files to better exploit the bandwidth.

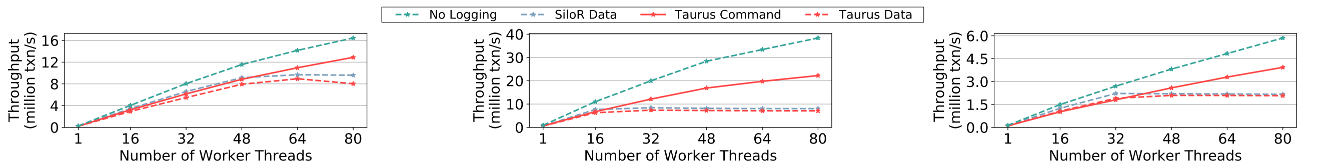
Logging Performance: Our first experiment evaluates the runtime performance of Taurus by measuring the throughput when the number of worker threads changes. We test the logging protocols with YCSB-500G and TPC-C benchmarks. We measure the throughput by the number of transactions committed by the worker 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 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 due to the disk bandwidth because the performance is similar on the RAID-0 disk array. It is instead because every transaction that spans multiple threads increments the shared LSN; this leads



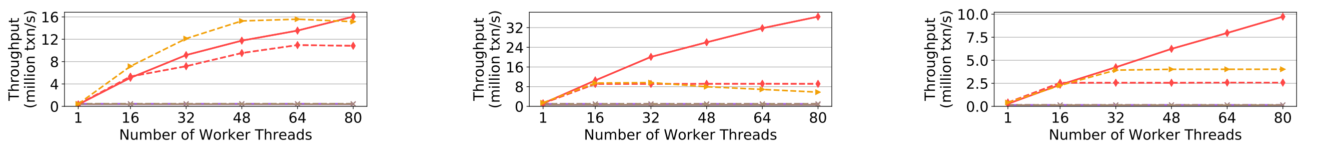
(a) YCSB-500G (b) TPC-C Payment (c) TPC-C New-Order

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



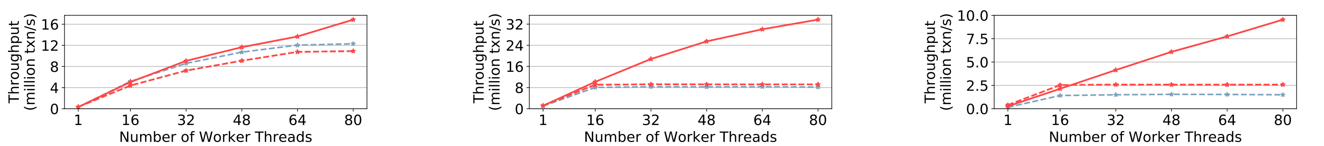
(a) YCSB-500G (b) TPC-C Payment (c) TPC-C New-Order

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



(a) YCSB-500G (b) TPC-C Payment (c) TPC-C New-Order

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



(a) YCSB-500G (b) TPC-C Payment (c) TPC-C New-Order

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

to excessive cache coherence traffic that inhibits scalability [35]. Taurus command logging is more scalable 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 each transaction, it generates a log record for each accessed partition, and accesses the per-log LSN to generate a global LSN for the transaction. Then, it 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-contended 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 takes 1.6% of the running time. Taurus command logging has the best performance. Plover suffers from the increased data accesses, causing the worker threads to compete for the latches on the local sequence numbers, essentially downgrading to a single stream logging. Fig. 5c shows the result for the New-Order transactions. These transactions access a larger number of tuples (~30 tuples each). The overall throughput is lower, making it difficult to hit the LSN allocation bottleneck. Therefore, serial command logging scales well. The gap between serial command with RAID-0 and Taurus command consists of LV-related overheads. Taurus shows advantages only when the number of workers is adequate. We project that the serial command logging will reach the cache traffic limit when there are 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. For all the benchmarks, both Silo-R and Taurus data logging plateau at a similar level, saturating the disk bandwidth. Before that, Silo-R performs slightly better than Taurus because it does not track LSN Vectors. However, Silo-R cannot track RAW dependencies, so it is incompatible with command logging. Taurus command outperforms Silo-R in every benchmark, by up to 2.8×.

Recovery Performance: We use the log files generated by 80 worker threads for better recovery parallelism. These files are large enough for steady measurements and are stored in uncompressed bytes across the disks with I/O caches cleaned. Fig. 7a shows the

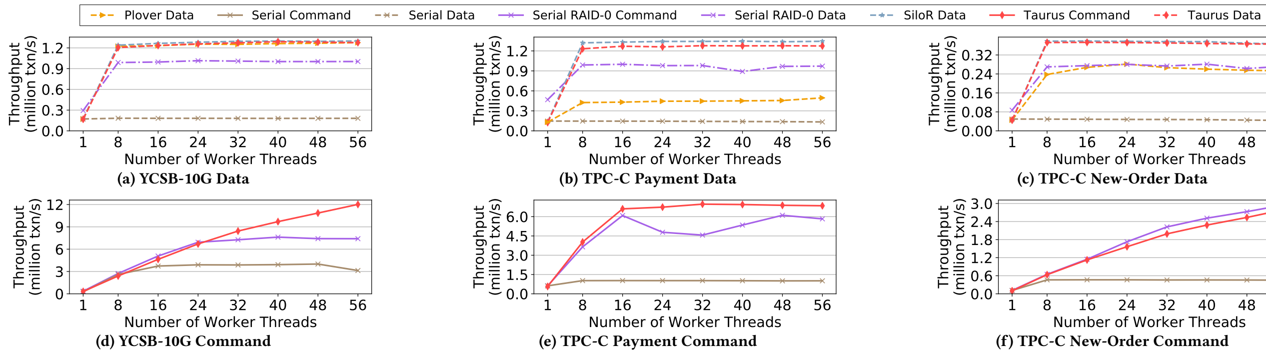


Figure 9: Data and Command Logging Performance comparison on YCSB-10G, TPC-C Payment, and TPC-C New-Order on HDDs.

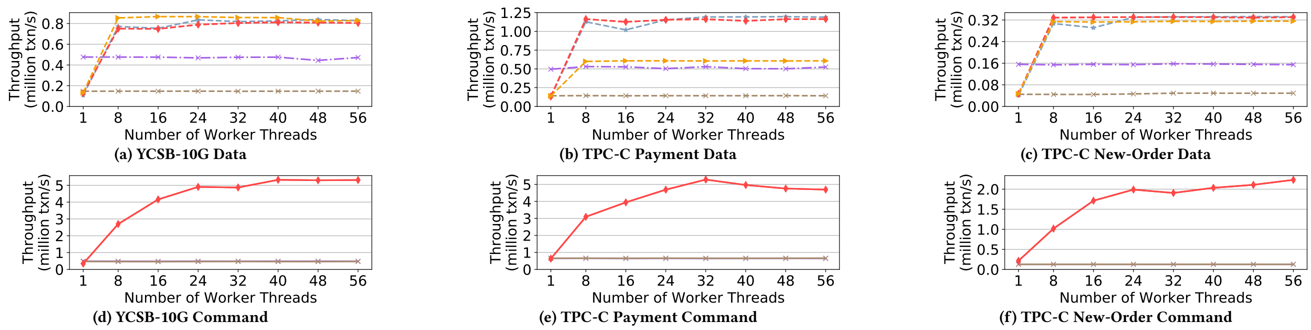


Figure 10: Data and Command Recovery Performance comparison on YCSB-10G, TPC-C Payment, and TPC-C New-Order on HDDs.

recovery peformance on YCSB-500G. Plover outperforms Taurus below 80 threads because it does not need to resolve dependencies. Each Plover log file contains totally ordered entries, sufficient to recover independently. Plover saturates the bandwidth after 48 threads. 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 is up to 42.6× faster than the serial baselines.

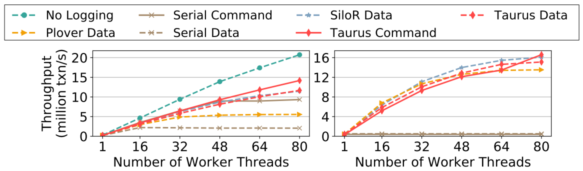
The recovery performance of TPC-C Payment is in Fig. 7b. Both Plover and Taurus data logging hit the I/O bottleneck quickly, while Taurus command logging scales linearly. Fig. 7c shows the comparison for TPC-C New-Order. Taurus command scales well and outperforms Plover by up to 2.4×. The gap between Plover and Taurus data is due 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 shows the results for the OCC baselines. Silo-R requires data logging and therefore falls behind Taurus command logging. But Silo-R does not need dependency resolution so it outperforms Taurus data logging when the number of transactions is large. Silo-R uses latches to ensure that transactions only apply updates with a higher version number. This overhead is more significant when transactions are long. Taurus command logging outperforms Silo-R by up to 9.7×.

5.3 Performance with Hard Disks

To better understand the performance of baselines with limited bandwidth, we evaluate them on an Amazon EC2 `h1.16xlarge` machine with eight HDDs. Each disk provides 160 MB/s bandwidth and in total the server has 1.3 GB/s bandwidth. Since the server only has 256 GB memory, we use YCSB-10G. The 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.

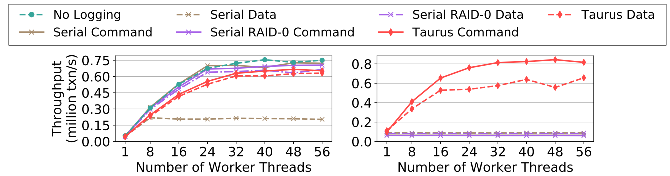
Logging Performance: Fig. 9a shows the logging performance of data logging baselines for YCSB-10G. We observe that serial data saturates the bandwidth of a single disk quickly. Taurus data logging achieves 7.1× higher throughput than serial data. Serial data logging on RAID-0 delivers similar performance since the bandwidth of the disk array is 8× greater. Silo-R and Plover also flush across eight disks uniformly, thereby achieving similar performance. 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 due to smaller log records. Starting from 16 threads, its performance is limited by the single disk bandwidth. 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 throughput for TPC-C Payment. Taurus plateaus after 16 threads, 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



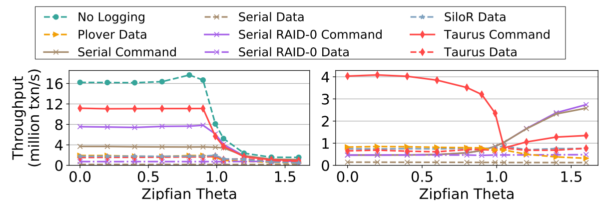
(a) YCSB-500G Logging (b) YCSB-500G Recovery

Figure 11: DRAM Performance.



(a) Logging (b) Recovery

Figure 12: TPC-C Full Mix Performance (2PL)



(a) YCSB-10G Logging (b) YCSB-10G Recovery

Figure 13: Contention. Varying Zipfian Theta in YCSB.

single socket. For the TPC-C New-Order workload in Fig. 9f, both serial command with RAID-0 and Taurus command scale well.

Recovery Performance: Fig. 10 shows the recovery performance. The serial baselines are again limited by the total order. For Taurus, 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 with RAID. Taurus data logging achieves similar throughput as Silo-R, while Taurus command logging is up to 6.3× faster. Plover parallels Silo-R except for Payment; here, it devolves into single-stream logging due to the contention.

The peak performance of Taurus command logging and Taurus data logging are 11.3× and 5.5× faster than the serial baselines for YCSB recovery. For TPC-C Payment in Fig. 10e, Taurus command logging is 7.1× faster than serial command logging. Its performance decreases with more than 24 workers because parallelism is fully exploited and more threads 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 instead of data logging, 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)

We evaluated the performance on DRAM filesystems to simulate a PM environment. Every operation to this filesystem goes through the OS. This overhead is shared in the real PM. The PM incurs a higher latency (<1 us for 99.99%) and has a bandwidth 3-13× lower compared to DRAM [40]. We conjecture that Taurus command logging would perform relatively better on a real PM because the bandwidth might become the bottleneck. Fig. 11 shows the results on the DRAM filesystem. The advantage of command logging is greatly reduced when the bandwidth is sufficient. Taurus command logging scales linearly, while serial command logging is restricted by the cache coherence traffic. All the parallel algorithms scale well in recovery. Silo-R outperforms Taurus slightly as it does not resolve dependencies. 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 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. Figure 12 shows the logging performance and recovery performance. Starting from 32 threads, the logging algorithms are limited by the workload parallelism. Compared to No Logging, the overhead caused by Taurus is around 11.7%. In recovery, Taurus command logging outperforms the serial baselines by 12.8×.

5.6 Sensitivity Study

Now we evaluate the performance when various factors change.

Contention: We use the YCSB-10G workload on the `h1.16xlarge` server to study how the contention level impacts performance. We adjust the θ parameter of the Zipfian distribution. A higher θ value corresponds to higher contention. Every baseline uses 56 workers.

Fig. 13a shows the throughput when varying θ for the logging. When θ is greater than 1.0, the performance of all the schemes decreases due to the reduced parallelism in the workload. Fig. 13b shows the recovery performance. It indicates different trends for serial algorithms and Taurus. For Taurus, the performance drops when θ goes beyond 0.8 due to the inter-log dependency issue (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 perform better with higher θ, because higher data skew makes the working set fit in on-chip caches, resulting in a higher cache hit rate and 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 is high (i.e., $\theta > 1$), we run Taurus with serial recovery to avoid the high latency between dependent transactions. This enables Taurus to achieve good performance under high contention.

Transaction Impact We evaluate YCSB-500G on an 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. When the number of tuples

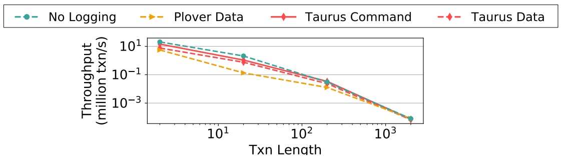


Figure 14: Transaction Impact. Varying the no. of accesses per txn.

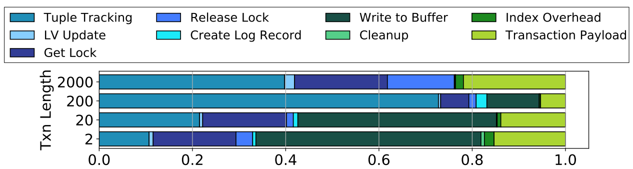


Figure 15: Transaction Impact. Time breakdown of Taurus Logging.

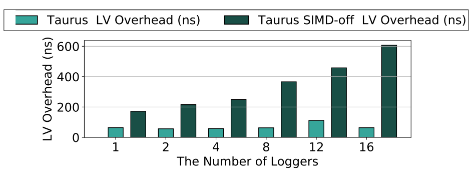


Figure 16: Vectorization. The LV Update overhead (in nanosec).

accessed per transaction increases from 2 to 200, the LV update overhead stays fixed at 0.6%, while the tuple tracking overhead of 2PL increases from 10.7% to 72.8%. With the NO_WAIT policy [43] to avoid 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 occur once per transaction, some like tuple tracking occur linearly in the number of tuples accessed, and some like getting the lock are more sensitive to the 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. Fig. 16 plots the time (in nanoseconds) of LV overhead per transaction with different numbers of log files. The gap increases with the number of log files. Using 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. Taurus includes ELR in its design.

Single-Storage Logging Algorithms: ARIES [29] has been the gold standard in database logging and is widely implemented. However, ARIES does not scale well on multicore processors, as many recent works have observed [16, 35, 37, 44]. C-ARIES [33] was proposed to support parallel recovery, and CTR [2] improves the recovery time by using multi-versioning and aggressive checkpointing, but the contention caused by the original ARIES logging remains.

Aether [16], ELEDA [18], and Border-Collie [20] have optimized ARIES by reducing the length of critical sections in logging. But they 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. These schemes 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 counter to order transactions, incurring scalability issues. Besides, the enforced order causes serial recovery. Adaptive logging [41] achieves parallel recovery for command logging in a distributed partitioned database. It infers dependency information from the transactions' read/write set. This approach maintains each transaction's start and end times to detect dependencies. PACMAN [38] enables parallel command logging recovery by using program analysis to learn what computation can be performed in parallel. Taurus supports both parallel logging and recovery, while [38] only supports parallel recovery.

Logging Algorithms for Modern Storage: Fast recovery based on NVM is an active research area [3, 4, 6, 10, 15, 21, 22, 36]. This line of work leverages the high bandwidth and byte-addressable nature of NVM to improve the performance. Taurus, in contrast, can work on both traditional HDD/SSDs and new NVM devices.

Dependency-Tracking Algorithms: Similar to Taurus, [8] also uses dependency tracking to log to multiple files, but does not log dependency information as metadata. This leads to two shortcomings: (1) transactions with dependencies have to be logged in order, incurring significant overhead when there are many inter-log dependencies; (2) it does not support parallel recovery. DistDGCC [42] is coupled with a dependency tracking logging scheme, but it logs fine-grained dependency graphs. In [17], Johnson et al. proposed a parallel logging scheme that relies on single-dimension Lamport clocks to achieve a global total order. 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 low. Taurus provides a serial fallback to fit low-parallelism cases.

Kuafu [14] is an algorithm for replaying transactions in parallel on a replica. It also encodes dependencies but only supports data logging. Bernstein et al. present a logging algorithm [5] for multi-partition databases. Their design uses two-dimensional vector clocks but keeps a total order among cross-partition transactions.

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.

Acknowledgments

This work was supported (in part) by the National Science Foundation (IIS-1846158, IIS-1718582, SPX-1822933, SPX-1822920), Google Research Grants, the Alfred P. Sloan Research Fellowship program, and Wisconsin Alumni Research Foundation.

References

[1] [n.d.]. DBx1000. https://github.com/yxymit/DBx1000.

[2] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. 2019. Constant time recovery in Azure SQL database. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2143–2154.

[3] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*. 707–722.

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

[5] Philip A Bernstein and Sudipto Das. 2015. Scaling Optimistic Concurrency Control by Approximately Partitioning the Certifier and Log. *IEEE Data Eng. Bull.* 38, 1 (2015), 32–49.

[6] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. 2015. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *VLDB* 8, 5 (2015), 497–508.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*. 143–154.

[8] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. 1984. *Implementation techniques for main memory database systems*. Vol. 14. ACM.

[9] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*. 1243–1254.

[10] Ru Fang, Hui-I Hsiao, Bin He, C Mohan, and Yun Wang. 2011. High performance database logging using storage class memory. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1221–1231.

[11] Colin J Fidge. 1987. *Timestamps in message-passing systems that preserve the partial ordering*. Australian National University. Department of Computer Science.

[12] Goetz Graefe and Harumi Kuno. 2016. Controlled lock violation for data transactions. US Patent 9,396,227.

[13] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. 2013. Controlled lock violation. In *SIGMOD*. ACM, 85–96.

[14] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. 2013. KuaFu: Closing the parallelism gap in database replication. In *ICDE*. IEEE, 1186–1195.

[15] Jian Huang, Karsten Schwan, and Moinuddin K Qureshi. 2014. NVRAM-aware logging in transaction systems. *VLDB* 8, 4 (2014), 389–400.

[16] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: a scalable approach to logging. *VLDB* 3, 1-2 (2010), 681–692.

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

[18] Hyungsoo Jung, Hyuck Han, and Sooyong Kang. 2017. Scalable database logging for multicores. *VLDB* 11, 2 (2017), 135–148.

[19] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (2008), 1496–1499.

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

[21] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in write-ahead logging. *ACM SIGOPS Operating Systems Review* 50, 2 (2016), 385–398.

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

[23] Hideaki Kimura, Goetz Graefe, and Harumi A Kuno. 2012. Efficient locking techniques for databases on modern hardware.. In *ADMS@ VLDB*. 1–12.

[24] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[25] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *VLDB* (2011), 298–309.

[26] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory OLTP recovery. In *ICDE*. 604–615. http://hstore.cs.brown.edu/papers/voltdb-recovery.pdf

[27] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. 215–226.

[28] Qingzhong Meng, Xuan Zhou, Shan Wang, Haiyan Huang, and Xiaoli Liu. 2018. A Twin-Buffer Scheme for High-Throughput Logging. In *International Conference on Database Systems for Advanced Applications*. 725–737.

[29] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[30] Yasuhiro Nakamura, Hideyuki Kawashima, and Osamu Tatebe. 2019. Integration of TicToc Concurrency Control Protocol with Parallel Write Ahead Logging Protocol. *International Journal of Networking and Computing* 9, 2 (2019), 339–353.

[31] David A Patterson, Garth Gibson, and Randy H Katz. 1988. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*. Vol. 17. ACM.

[32] Eljas Soisalon-Soininen and Tatu Ylönen. 1995. Partial strictness in two-phase locking. In *International Conference on Database Theory*. Springer, 139–147.

[33] Jayson Speer and Markus Kirchberg. 2007. C-ARIES: A multi-threaded version of the ARIES recovery algorithm. In *International Conference on Database and Expert Systems Applications*. Springer, 319–328.

[34] The Transaction Processing Council. 2007. TPC-C Benchmark (Revision 5.9.0).

[35] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *SOSP*.

[36] Tianzheng Wang and Ryan Johnson. 2014. Scalable logging through emerging non-volatile memory. *VLDB* 7, 10 (2014), 865–876.

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

[38] Yingjun Wu, Wentian Guo, Chee-Yong Chan, and Kian-Lee Tan. 2017. Fast Failure Recovery for Main-Memory DBMSs on Multicores. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 267–281.

[39] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. 2020. Taurus: Lightweight Parallel Logging for In-Memory Database Management Systems (Extended Version). arXiv:arXiv:2010.06760

[40] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 169–182.

[41] Chang Yao, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, and Sai Wu. 2016. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *SIGMOD*. 1119–1134.

[42] Chang Yao, Meihui Zhang, Qian Lin, Beng Chin Ooi, and Jiatao Xu. 2018. Scaling distributed transaction processing and recovery based on dependency logging. *VLDB Journal* 27, 3 (2018), 347–368.

[43] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *VLDB*, 209–220.

[44] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*. 465–477.

[45] Huan Zhou, Jinwei Guo, Huiqi Hu, Weining Qian, Xuan Zhou, and Aoying Zhou. 2020. Plover: parallel logging for replication systems. *Frontiers of Computer Science* 14, 4 (2020), 144606.