Taurus: A Parallel Transaction Recovery Method Based on Fine-Granularity Dependency Tracking

Xiangyao Yu
CSAIL MIT
yxy@csail.mit.edu

Siye Zhu
Phillips Academy
szhu@andover.edu

Justin Kaashoek
Lexington High School
justin.kaashoek@gmail.com

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Srinivas Devadas
CSAIL MIT
devadas@csail.mit.edu

ABSTRACT

Logging is crucial to performance in modern multicore main-memory database management systems (DBMSs). Traditional data logging (ARIES) and command logging algorithms enforce a sequential order among log records using a global log sequence number (LSN). Log flushing and recovery after a crash are both performed in the LSN order. This serialization of transaction logging and recovery can limit the system performance at high core count.

In this paper, we propose Taurus to break the LSN abstraction and enable parallel logging and recovery by tracking fine-grained dependencies among transactions. The dependency tracking lends Taurus three salient features. (1) Taurus decouples the transaction logging order with commit order and allows transactions to be flushed to persistent storage in parallel independently. Transactions that are persistent before commit can be discovered and ignored by the recovery algorithm using the logged dependency information. (2) Taurus can leverage multiple persistent devices for logging. (3) Taurus can leverage multiple devices and multiple worker threads for parallel recovery. Taurus improves logging and recovery parallelism for both data and command logging.

Our evaluation on a 32-core machine with four persistent devices shows that Taurus can improve the data (command) logging performance by up to 3.8× (2.67×) and reduce recovery time by 4.9× (9.8×) compared to baseline algorithms.

1. INTRODUCTION

Logging is an important component for on-line transaction processing (OLTP) database management systems (DBMSs). It enables the system to guarantee that if a transaction commits that its changes are persistent and recoverable after a system crash. The changes are written to persistent storage (e.g., HDD, SSD, NVM) as an append-only log that contains enough information to restore the database to a consistent state as it existed before the crash.

ARIES [19] is the most popular logging algorithm and has been widely implemented in both disk-based and in-memory database systems. Command logging [18] was recently proposed to reduce the amount of log data by logging transactions commands instead of actual modifications. In both ARIES and command logging, each transaction acquires a unique and monotonically increasing *log sequence number* (LSN) that determines the order that the DBMS flushes log records to persistent storage. All the transactions need to be written to persistent storage sequentially in a single stream.

But the LSN allocation and the single I/O stream become a performance bottleneck when executing highly concurrent OLTP workloads on a DBMS with multiple CPU cores [13]. Furthermore, during crash recovery, replaying logs the LSN order limits the level of parallelism. We observe that using a global LSN to determine the order of logging is not necessary. If two transactions access completely disjoint set of data, in theory, they can perform logging and recovery in parallel (potentially to/from different log files) without communicating with each other. Which transaction becomes durable and commits first does not affect the recoverability of the system. In serial logging algorithms, however, these two transactions will be assigned different LSNs that unnecessarily serialize the logging process.

To overcome this artificial bottleneck, we propose **Taurus**, a logging algorithm that can perform both logging and recovery in parallel. Taurus removes the global LSN bottleneck by explicitly maintaining dependency information between different transactions. And the *dependency information is written to log records*. This provides two major benefits. First, the commit and recovery order is enforced only for transactions with true dependency but not for independent transactions. For workloads with ample inherent parallelism, Taurus is able to perform both logging and recovery in parallel. Second, the logging and commit processes are decoupled. A transaction can write to a log in any order with respect to its predecessor transactions; the dependency information is able to determine the proper commit and recovery order regardless of the logging order.

A key idea behind Taurus that makes the dependency tracking efficient is that only *read-after-write* (RAW) and *write-after-write* (WAW) dependencies need to be tracked but not the *write-after-read* (WAR) dependencies. Therefore, it suffices to only maintain the last writer of each object (to detect RAW and WAW) but not the readers. This simplifies the dependency tracking during logging and recovery.

Recent work has also tried to improve the scalability of logging. Some focused on optimizing ARIES but still suffer from the LSN allocation bottleneck [13]; others allocate LSN at coarse temporal granularity and perform logging in batches to trade latency for throughput [23, 27, 24]. Taurus is unique as it breaks the LSN abstraction to track dependency graph explicitly at a fine granularity. Taurus is completely compatible with other design dimensions. For

example, it parallelize the logging and recovery for both data and command logging.

To evaluate our approach, we implemented our Taurus logging scheme in the DBx1000 [2] open source database. We compare our methods with other state-of-the-art approaches and show that they enable the DBMS to achieve up to 4.9× higher throughput across different OLTP workloads. We also demonstrate that Taurus allows the DBMS to reduce its recovery time by 9.9× for command logging.

The remainder of this paper is organized as follows. We first provide an overview of logging in OLTP DBMSs and our target operating environment in Section 2. Next, in Section 3, we describe the key idea of Taurus, the dependency tracking. Section 4 explains the details of Taurus with data logging. Section 5 extends the Taurus to command logging. We present our experimental evaluation in Section 6, discuss the related work in Section 7 and conclude the paper in Section 8.

2. BACKGROUND

A transaction is a sequence of actions that executes on a shared database to perform some higher-level function. Ideally, an OLTP DBMS provides the ACID guarantees for transactions [12]. Atomicity guarantees that a transaction either succeeds or fails, leaving the database unchanged if it does fail. Consistency implies that every transaction will bring the database from one valid state to another and isolation ensures that the result obtained from a set of concurrent transactions will be the same as if those transactions were executed serially.

Durability is also essential to a system because it guarantees that a committed transaction can be recovered after a system crash. In this paper, we consider main memory DBMSs which do not store database states on persistent storage besides the log. The original ARIES algorithm can be significantly simplified in this setting [18].

To simplify the discussion, we define some terminologies and make a few assumptions of the DBMS. We assume it is a main memory database with all uncommitted data staying in DRAM. Therefore, a transaction only needs REDO data but not UNDO data. We further assume that each transaction only creates a single log record which is flushed to persistent storage at the end of its execution. Log records are sequentially flushed to storage devices in batches. The machine contains multiple storage drives that it can write logs to.

The execution of a transaction contains two phases. In the *execution* phase, it reads and writes tuples and executes the transaction logic. For simplicity, we only consider serializable isolation in this paper. If the transaction is successfully executed, it will pre-commit meaning that the transaction will not abort unless the system crashes [21]. However, the transaction can only commit (i.e., be returned to the end users) after it becomes committable. Committability and the theory of durability will be discussed in the next section.

2.1 Durability Theory

The theory of recoverability was first established as part of the *serializability theory* by Bernstein et al [6]. According to the theory, a history is recoverable if each transaction commits after the commitment of all transactions (other than itself) from which it reads. Essentially, this requires the commit order of transactions to follow the Read-After-Write (RAW) dependencies. However, other types of dependencies, Write-After-Read (WAR) and Write-After-Write (WAW), do not have any constraint on the commit order. We can therefore define the committability of a transaction.

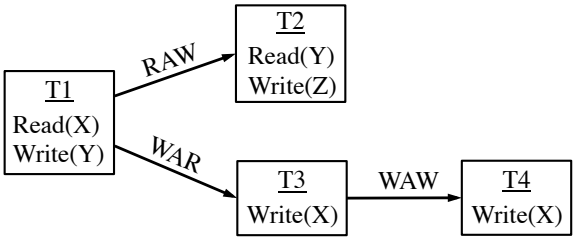


Figure 1: A example of a set of transactions with different types of data dependencies.

DEFINITION 1 (COMMITTABLE TRANSACTION). *A transaction is committable if (1) all of objects that it reads were updated by transactions that are committable and (2) its REDO log is persistent.*

A transaction can commit once it is committable. During recovery after a crash, *all and only* committable transactions should be recovered. A correct recovery algorithm should identify all transactions that successfully committed before the crash, and recover them in the proper order such that the recovered state is the same as that before the crash.

Figure 1 shows an example of a set of four transactions (T1 to T4) executed in a DBMS. The types of data dependency between different transactions are shown as arrows. For simplicity, we assume that all the transactions have pre-committed but none is persistent and therefore none has committed yet.

In the example, T1 is committable after its log record is written to persistent storage. For T2, according to Definition 1, it needs to wait for T1 to be committable before it can commit due to the RAW dependency between T1 and T2. If we ignore this constraint and commit T2 before T1 is persistent, then if the system crashes, only T2 will be recovered but T1 will not. This leads to inconsistent state as T2's read(Y) observes a value that does not exist in the recovered system.

Unlike RAW dependency, WAR dependency does not constrain the commit order. T3 in Figure 1, for example, can commit before T1 regardless of the WAR dependency between them. If a crash happens and T3 is recovered but T1 is not, the recovered system is still in a consistent state as if T1 has never existed. This is correct because a read does not leave side effects in the system, and therefore does not affect a following write.

Similarly, WAW dependency does not constrain the commit order either. A WAW dependency without a RAW means the second transaction blindly overwrites the value without reading it first. In the example, this means that T4 completely overwrites the value of X previously written by T3. According to Definition 1, T4 can commit before T3. If after a crash only T4 is recovered but T3 is not, the database state is perfectly consistent. As if only T4 has been executed and T3 has never existed.

Note that the concurrency control algorithm has stronger requirement than the logging algorithm, where all the three types of data dependencies need to be taken care of. We will leverage this relaxed requirement in the design of Taurus.

We observed that some previous work on database logging ignored the fact that WAR and WAW dependencies do not enforce commit order [21, 24, 23, 13, 10]. Therefore, all three types of dependencies were treated the same way. We will show later how ignoring WAR and WAW can substantially improve the scalability of database logging and recovery.

2.2 Serial Data and Command Logging

Traditional logging algorithms like ARIES log the modification made by a transaction as REDO information to persistent storage.

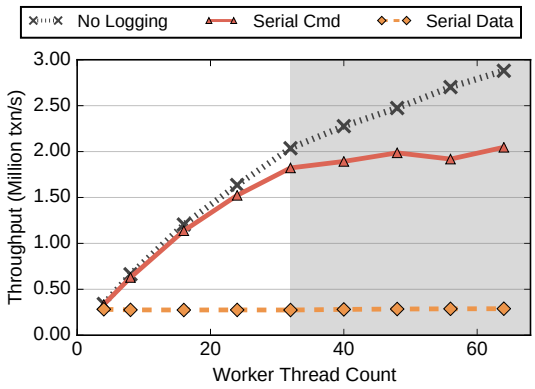


Figure 2: Logging and recovery performance for serial data and command logging on TPCC workload (32 warehouses).

For transactions that modify a large number of tuples, this method will generate a large amount log data leading to high pressure at the I/O. As shown in Figure 2, the performance of serial data logging can saturate at very low thread count due to the I/O bottleneck.

Command logging was recently proposed [18] to mitigate the I/O pressure during the forward execution phase of a DBMS. Instead of logging the actual modification made by a transaction, the database logs the command of the transaction instead, which includes the store procedure name and the input parameters. Compared to data logging, command logging can reduce the size of logs by an order of magnitude, therefore significantly mitigate the I/O pressure. As can be seen in Figure 2, command logging significantly improves the performance during forward processing. The throughput, however, still saturates at high thread count when the I/O becomes the bottleneck.

In both serial data and command logging algorithms, a transaction is assigned a unique *Log Sequence Number* (LSN). The LSN order is the global serial order of transactions and therefore is in accordance with all data dependencies. Namely, if T1 depends on T2 (regardless of RAW, WAW and WAR), T1's LSN must be greater than T2's. This property can be achieved through the concurrency control algorithms. For examples, two-phase locking algorithms would allocate the LSN after precommitting but before releasing any locks, while timestamp ordering algorithms can use the logical commit timestamp as the LSN.

Transactions using serial logging algorithms must flush their log records to persistent storage in the LSN order. According to Definition 1, a transaction becomes committable after flushing its log record, because all transactions that it RAW depends on must have been flushed as well. During a crash recovery, all complete persistent log records must have been committed and therefore should be recovered. The recovery will following the LSN order, making sure that updates from conflicting transactions are applied in the correct sequence.

A major scalability bottlenecks in serial logging algorithms are the LSN allocation and the serial log writing. For modern multicore systems where a large number of transactions can commit at the same time, these bottlenecks will significantly limit performance. We observe, however, that LSN overly constraining the commit order of transactions. According to Definition 1, one prerequisite of committability is that *"all transactions **it reads from** are committable"*. In ARIES, however, this prerequisite becomes that *"all transactions **before it in the serial order** are committable"*. Clearly, the commit and recovery order constraint in serial logging is much stronger than what is actually required.

3. TAURUS DEPENDENCY TRACKING

We propose Taurus to improve the efficiency and scalability of database logging. Taurus is able to exploit mulitple persistent storage devices by logging to them in parallel. It can also exploit multiple recovery threads to replay transactions in parallel. Taurus achieves this by tracking down *and log* the dependency between different transactions and enforce the commit and recovery order based on such dependency. Taurus maximize parallelism for both data and command logging schemes.

A key feature of Taurus is that it decouples transactions' logging with their commit. In Taurus, a transaction can write to any logger and different loggers can flush to persistent storage in any order. This provides great parallelism and flexibility as no logging order needs to be enforced. However, this means that a transaction T may become persistent before its predecessors which is not allowed in traditional logging algorithms. Therefore, *not all persistent transactions* have committed and a separate software module is required to determine the committability of each transaction. After a system crash, the recovery process needs to identify transactions that haven't committed before the crash and ignore them for recovery. Taurus achieves these using the logged dependency information in each transaction.

In Taurus, a transaction collects its RAW and WAW predecessors (which are the IDs of last writing transactions of accessed tuples) during normal execution. After a transaction pre-commits, it picks a logger and writes its log record which contains the REDO data as well as the RAW and WAW predecessor list. A transaction can commit only if the two conditions in Definition 1 are met. Namely, all RAW predecessors are committable (condition 1) and its REDO data has been persistent (condition 2). Checking the second condition is straightforward, the transaction simply wait for log file flush. Checking the first condition, however, requires a transaction to check committability of transactions it depends on. We will discuss more details of the committability check in Section 4.2.

During a crash recovery, Taurus constructs the *recovery dependency graph* using the predecessor list from each log record. The transactions are recovered in a dataflow fashion following the graph. Taurus determines that a transaction T has committed (i.e., recoverable) if all its RAW predecessors are recoverable. Unrecoverable transactions are ignored. For data logging, a recoverable transaction starts recovery after all its WAW predecessors have been recovered. For command logging, recovery starts when both RAW and WAW predecessors of a transaction are recovered. Independent transaction can be recovered in parallel.

Specifically, if a RAW predecessor of a transaction T is not found in any log file, the predecessor must have not committed before the crash and therefore T has not committed either according to Definition 1. In this case, T and all RAW successors of T are not recoverable and ignored by Taurus. In contrast, if a WAW predecessor of T is not found in any log file but all RAW predecessors have been found and recovered, T can still be recovered. The fact that a WAW predecessor did not commit does not affect its committability and recoverability.

3.1 Example

Figure 3 shows the log records of transactions in the example of Figure 1. A log record contains both the REDO information and the predecessor list (if any). During normal execution, if the log record of T1, T3, or T4 is flushed to any logger, they can immediately commit and return to the end users. This is because they do not RAW depend on any other transactions and condition 2 in Definition 1 is always true.

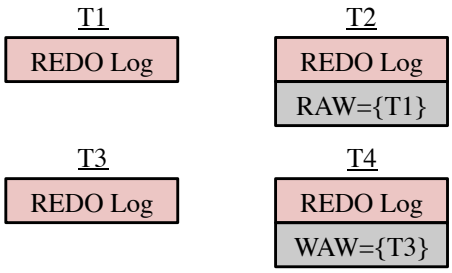


Figure 3: Taurus log records for transactions in Figure 1. A log record contains both REDO data and predecessor list.

To commit T2, however, we also needs to check that T1 has committed because of the RAW dependency. But as discussed above, this only limits their commit order but not the logging order. T1 and T2 can log to any logger in any order. The commit dependency logic guarantees that T2 commits after T1 is committable regardless of their logging order.

During recovery, Taurus first uses RAW edge to determine recoverability of each transaction. T1, T3 and T4 are always recoverable due to the lack of RAW predecessors. But T2 is recoverable only if T1 is recoverable. For data logging, the actual recovery follows the WAW edges. T1, T2 and T3 can be recovered in parallel. But T4 should be recovered after T3. For command logging, T1 and T3 will be first recovered in parallel, and then T2 and T4 can be recovered.

4. DATA LOGGING IN TAURUS

In this section, we focus on the discussion of data logging and recovery of Taurus. Command logging and recovery will be discussed in Section 5. Specifically, we first explain data structures required in Taurus's implementation in Section 4.1. We explain the committability test in Section 4.2 and the full logging algorithm in Section 4.3.

4.1 Data Structures

In order to track dependency between transactions, the following data structures are added/modified in Taurus.

4.1.1 Tuple

A data tuple consists of the stored data and concurrency control information (assuming tuple level concurrency management). In Taurus, a *last writer* field is added to each tuple to detect RAW and WAW dependencies. The *last writer* is the ID of the last transaction in the system writing to the data tuple. It is updated when the tuple is modified. A transaction maintains lists of RAW and WAW predecessors in its local storage.

Note that a *last writer* field is enough for RAW and WAW dependency tracking but not enough for WAR dependencies. As discussed in Section 2.1, however, WAR does not need to be tracked for logging. This is the key to the efficiency of parallel logging in Taurus.

4.1.2 Log Record

In our execution model, the system generates a log record at the end of each transaction. The log record consists of the following fields.

Txn_id. The transaction ID is the unique identification information of each transaction in the system. It is similar to LSN in ARIES, but it does not have to reflect the global sequential order. In this paper, it is the address of a transaction in a log file concatenated with the logger ID.

REDO data. This field stores the necessary information that allows the system to recover a transaction after a crash. It is the same information as in traditional logging algorithms. In data logging, it includes the tables ids, primary keys, before and after images, etc. In command logging, it includes the store procedure name and input parameters required to rerun the transaction.

Predecessors. This field contains the `last_writers` of all tuples read/written by the current transaction. It consists of RAW and WAW predecessors separately. From the discussion in the previous section, only RAW predecessors are necessary to determine recoverability, but WAW predecessors are also needed to determine the recovery order.

4.1.3 Commit Dependency Table

We use the *commit dependency table* to determine the committability of each transaction. The table contains transactions that have pre-committed but not committed yet. For each transaction in the table, it stores a *predecessor vector* which is a vector of addresses of all loggers that have to be flushed before the transaction is committable. A predecessor vector of (100, 200), for example, means that the transaction is committable if address up to 100 is persistent in the first logger and address up to 200 is persistent in the second logger. An entry is inserted to the table when a transaction pre-commits and removed when it commits. The predecessor vector will propagate among dependent transactions and the detailed algorithm will be explained in Section 4.2.

4.1.4 Recovery Dependency Graph

Taurus supports parallel recovery of independent transactions. The *recovery dependency graph* maintains the pairwise dependencies between transactions. A transaction is recoverable if all its RAW predecessors are also recoverable. For data logging, a transaction starts recovery after all WAW predecessors are recovered.

We use nodes to represent transactions and edges directed from a transaction to its dependents to indicate the pairwise dependencies. Having edges in this direction allows a recovered transaction to quickly notify its dependents following the edges. It is also possible to have edges pointing to the other direction. With this alternative design, a recovered transaction cannot notify dependents; and the dependents need to periodically check whether a predecessor has been recovered or not, which would incur more computation and latency.

The recovery dependency graph is constructed and maintained in main memory. Our current implementation stores the graph in a hash table which is very scalable as operations to different buckets can be performed in parallel. But any data structure supporting insert, delete and lookup will work. Each node in the recovery dependency graph contains the following fields.

REDO data. The necessary information to recover the transaction. This field is copied from the REDO data field in the log record.

Pred_size. A node maintains both *raw_pred_size* and *waw_pred_size*. *raw_pred_size* is the number of unrecoverable RAW predecessors and *waw_pred_size* is the number of unrecovered WAW predecessors. *raw_pred_size* (or *waw_pred_size*) is decremented when a RAW (or WAW) predecessor becomes recoverable (or be recovered). A transaction is recoverable if *raw_pred_size* is 0. After being recoverable, it can start recovery if *waw_pred_size* becomes 0.

Successors. *raw_successors* and *waw_successors* are lists of the IDs of the transactions that RAW and WAW depend on the current transaction, respectively. When a transaction is inserted to the graph, it is responsible for adding itself to all predecessors' *successors* list.

When a transaction changes it state, it is responsible to notify all transactions in its corresponding *successors* list by decrementing their *pred_size*.

4.2 Committability Test

A transaction is committable when the two conditions in Definition 1 are true. The challenge is to enforce the first condition, namely, all transactions it reads from are committable. We want to enforce it without serializing the commit order of all transactions as done in serial logging algorithms.

A generic solution is to maintain all the transactions in a dependency graph where each node represents a transaction and each edge represents a RAW relationship. A transaction can commit after all its RAW predecessors commit. The checking of different transactions can be performed in parallel and independent transactions can commit in parallel.

Although this solution is viable and provides more parallelism than serial logging, maintaining the dependency graph explicitly can be expensive, adding overhead to the logging process. For workloads that generate small amount of log records where a single log stream performs well, the extra complexity of dependency tracking can hurt performance.

We observe, however, that the dependency graph can be largely compressed by leveraging the fact that a log file is flushed to persistent storage sequentially. Therefore, if a record becomes persistent, all records before it in the same log file must be persistent as well. In Taurus, instead of tracking the whole dependency graph, we only track the position in each log file that has to be persistent for a transaction to commit. We use a per-tuple *predecessor vector* (similar to version vector [20]) to achieve this goal.

If a transaction T RAW depends on a list of other transactions, T calculate its predecessor vector by taking the maximum of all predecessors' predecessor vector and the position of its own log record. The maximum of two predecessor vector is just their component-wise maximum. Each logger maintains a buffer in main memory which is flushed to persistent storage sequentially. A logger's *persistent point* is the position in the logging stream up to which has been persistent already. Each persistent point only moves up in the buffer. The *persistent vector* is the vector of all loggers' persistent point. A transaction is committable if the persistent vector is greater than its predecessor vector.

Commit Dependency Table

Transaction ID	Predecessor Vector
1	(100, 0)
2	(100, 200)
3	(100, 300)
…	…

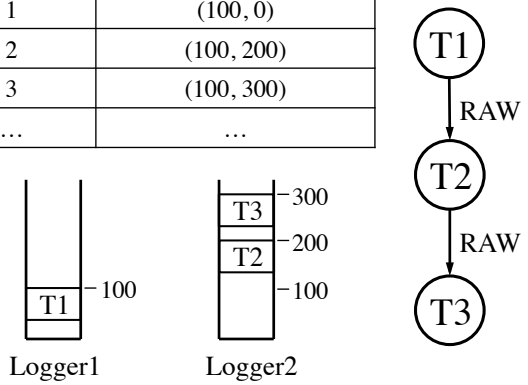


Figure 4: Determine committability of a transaction in Taurus.

Figure 4 shows an example of three transactions where T2 RAW depends on T1 and T3 RAW depends on T2. We have two loggers in this example. When T1 pre-commits, it allocates an entry in logger 1 to position 100. Since T1 has no RAW predecessors, its predecessor vector is (100, 0). The vector is written to the commit dependency table. T1 is committable when Logger 1's persistent point reaches 100, in other words, the persistent vector is greater than (100, 0).

When T2 reaches the pre-commit point, it computes its own predecessor vector. Because it allocates an log entry in logger 2 to position 200, its predecessor vector is no less than (0, 200). Also since it RAW depends on T1, it looks for T1 in the commit dependency table. If the entry is found (i.e., T1 has not committed yet), T2 updates its predecessor vector to be the maximum of its current predecessor vector and T1's predecessor vector, namely (100, 200) = *Max*((0, 200), (100, 0)). This is inserted to the commit dependency table for T2. Similarly, the predecessor vector of $T3$ will be computed as (100, 300) since it depends on T2 and its log record is in logger 2 at address 300.

Note that it is possible that T1 commits before T2 or T3 pre-commits. In this case, T1 will be removed from the commit dependency table and following dependent transactions will ignore it when computing their predecessor vector. However, since T1 has already committed, persistent vector is already greater than its predecessor vector and therefore there is no need to keep it any more.

Predecessor vectors simplify testing of committability of transactions. A transaction is committable if and only if the persistent vector is no less than the predecessor vector of the transaction. If the persistent vector is (100, 250), for example, T1 and T2 are able to commit while T3 is not. No dependency graph needs to be explicitly tracked with this approach.

4.3 Parallel Logging Algorithm

In this section, we discuss how Taurus performs parallel logging during normal execution. The recovery algorithms after a system crash will be discussed in Section 4.4.

When a transaction T accesses a tuple during its execution, besides the data reads/writes and concurrency control metadata management, the routine shown in Algorithm 1 is executed. The `last_writer` of each data tuple accessed by T is recorded in *raw_predecessors* or *waw_predecessors* for a read or write respectively.

Algorithm 1 Transaction T accesses tuple.

```
1: function ACCESS(type, tuple)
2:     if type = Read then
3:         raw_predecessors.push(tuple.last_writer)
4:     else if type = Write then
5:         waw_predecessors.push(tuple.last_writer)
6:     end if
7: end function
```

If transaction T reaches the pre-commit point, the logging process shown in Algorithm 2 is executed. We first updates the *last_writer* fields of all the tuples updated by T (line 1-3). Then, a log record is created and written to a logger (line 4-5). The logger thread will flush the records to persistent storage in the background. In Taurus, any transaction can write to any logger without affecting correctness. In our implementation, we use a static mapping between worker threads and loggers.

The predecessor vector management is handled in line 6-13 where T updates its pred_vector based on its own log record position as well as its RAW predecessors. T is able to pre-commit and release

locks after that (line 14). Finally, when the persistent vector is greater than or equal to a transaction's predecessor vector, the transaction can be committed and removed from the commit dependency table (line 15-20).

Algorithm 2 Logging routine for transaction T.

```
    # When Transaction T is able to pre-commit
 1: for all t in T.WriteSet do
 2:     t.last_writer ← T.id
 3: end for
 4: log_record ← T.createLogRecord()
    # T's log entry is written to position addr in a logger
 5: addr ← loggers[logger_id].write(log_record)
 6: pred_vector ← [0] × num_loggers
 7: pred_vector[logger_id] = addr
    # CDT: Commit Dependency Table
 8: for all id in T.raw_predecessors do
 9:     if CDT.find(id) then
10:         pred_vector ← Max(pred_vector, CDT[id])
11:     end if
12: end for
13: CDT.insert(T.id, pred_vector)
14: T.preCommit() # e.g., release locks
15: pending_queue.push(T)
16: while pending_queue.head.pred_vector ≤ persistent_vector do
17:     T ← pending_queue.pop()
18:     T.commit()
19:     CDT.remove(T.id)
20: end while
```

In our current implementation, pending transactions that have pre-committed but not committed yet are stored in a thread local queue. And the thread is responsible for committing these transactions once they are committable. We chose a queue-based implementation for simplicity, more sophisticated data structures might allow pending transactions to be flushed out of order leading to potentially better performance.

4.4 Parallel Recovery Algorithm

According to our discussion in Section 4.2, a correct recovery process should (1) recover all and only transactions that were committable before the crash (i.e., recoverable transactions), and (2) recover transactions in the same logical order in which they were originally executed.

In traditional logging algorithms like ARIES, the recoverability check is trivial since all persistent transactions were committable before the crash. Taurus, however, allows non-committable transactions to be logged first. Therefore, Taurus requires non-trivial recoverability check. Using the dependency information logged in each transaction, Taurus can perform both the recoverability check and the recovery process in parallel. The parallelism comes from two factors. First, Taurus is able to read from multiple log files in parallel. Second, Taurus is able process independent transactions using different worker threads in parallel.

In Taurus, we use the RAW predecessors in each log record to decide recoverability of a transaction; and use the WAW predecessors to decide the recovery order. We maintain the dependency information in a graph called the *recovery dependency graph* and determine recoverability and recovery order using it. In Section 4.4.1, we present a basic parallel recovery algorithm for data logging. We then propose optimizations to it for better performance in Section 4.4.2.

4.4.1 Basic Parallel Recovery

A simple way of implementing parallel recovery is to break the algorithm into three phases to construct the graph, determine recoverability and then recover transactions. The system first recovers the latest checkpoint. Then the logging threads read log records from the files and do some preprocessing. The recoverability check and transaction recovery are performed by worker threads. Specifically, each phase performs the following logic.

Phase 1 (Construct dependency graph). The logging threads scan log files in parallel and construct the recovery dependency graph using the RAW and WAW predecessor information in each log record.

Phase 2 (Remove unrecoverable transactions). Since not all persistent transactions are recoverable in Taurus, we have to identify and discard unrecoverable ones. This is done by scanning the recovery dependency graph following RAW edges and remove all non-reachable transactions, i.e., transactions with some RAW predecessors not found in the log files. These transactions did not commit before the crash and therefore are unrecoverable.

Phase 3 (Recover transactions). Similar to Phase 2, worker threads scan the recovery dependency graph again but follows WAW edges this time. A transaction can start recovery if all its WAW predecessors have been recovered.

We use the example in Figure 1 to illustrate the mechanism of our algorithm. We consider the case where all the four transactions are persistent (and therefore committable) before the crash.

After running phase 1, the dependency graph shown in Figure 1 will be constructed (except the WAR edge between T3 and T1). The second phase first marks T1, T3 and T4 as recoverable, since they have no RAW predecessors. Then T2 is also marked as recoverable following its RAW predecessors (i.e., T1). The third phase recovers T1, T2, and T3 in parallel and then recovers T4 after T3 is recovered.

Another interesting case to consider is that only T2 and T4 are found in the log files. In this case, phase 1 only constructs the graph with T2 and T4. During Phase 2, T2 will be discarded since its RAW predecessor (i.e., T1) is not found in any log file. T4, however, is recoverable since it has no RAW predecessor. Finally in the third phase, T4 will be recovered. The fact that T3 is not found does not affect its recovery.

4.4.2 Optimized Parallel Recovery

One downside of the basic recovery algorithm discussed in the previous section is the size of the recovery dependency graph. The graph contains all transactions in the log files since the last checkpoint, and therefore can occupy significant DRAM capacity leading to performance degradation. We observe, however, that it is not necessary to construct the whole graph before recovering the transactions. We can merge the three phases in the previous section and process Phase 2 and 3 as the graph is constructed, and garbage collect nodes from the graph as transactions get recovered. This way, the graph can remain small which leads to better cache hit rate and higher performance.

The optimized parallel recovery algorithm is shown in Algorithm 3. Specifically, the logging threads read from the log files and insert records to the recovery dependency graph (line 1-2). The logging threads also handle garbage collection of recovered nodes (line 3). All transactions that are ready for recovery are inserted into the *recover_queue*. Transactions in the queue do not have WAW conflicts with each other and thus can be recovered in any order. The worker threads grab transactions from the queue and recovery them (line 6-8). Finally, a recovered transaction notifies its WAW dependents (line 9-15). If any WAW dependent has no pending predecessors, it is inserted to the *recovery_queue*.

Algorithm 3 Recovery routine for transaction T.

```
    # I/O Thread
 1: while R ← readNextLogRecord() do
 2:     recovery_dependency_graph.Insert(R)
 3:     GarbageCollect(R)
 4: end while

    # Worker Thread
 5: while recoveryNotDone() do
        # recover_queue keeps transactions ready for recovery
 6:     if R ← recover_queue.pop() then
 7:         recover(R.redo_info)
 8:         R.state ← Recovered
 9:         for all S in R.node.waw_successors do
10:             S.waw_pred_size −−
11:             if S.raw_pred_size = S.waw_pred_size = 0 then
12:                 N.state ← ReadyForRecovery
13:                 recover_queue.push(S)
14:             end if
15:         end for
16:     end if
17: end while
```

For the rest of this section, we discuss in details how each part of Algorithm 3 is implemented in Taurus. Each transaction is in one of four states during recovery. *Unrecoverable* means the transaction has some unrecoverable RAW predecessors (e.g., not read from the log files yet); newly inserted transactions are in this state. *Recoverable* means all RAW predecessors of the transaction are recoverable, but some WAW predecessors have not been recovered yet. These transactions will be recovered eventually after its WAW predecessors. *ReadyForRecovery* means the transaction is recoverable and all WAW predecessors are also recovered; these transactions are put into the recover_queue and can start recovery at any time. Finally, *Recovered* means the transaction has already been recovered and therefore can be considered for garbage collection. These four states correspond to where a transaction is in the three phases in Section 4.4.1. In an actual implementation, the four states do not need to be explicitly maintained. But we have them in the algorithms to help understanding.

The I/O threads construct the recovery dependency graph by calling the Insert() method shown in Algorithm 4. All the nodes in the graph are stored in *Nodes* which is a lookup table. We implemented it using a hash table but any data structure supporting lookup, insert and delete should work. A node is created for the transaction if it does not exist already (line 2-7). The redo_info is copied from the log record to the node (line 8). The algorithm than goes through all the predecessors of the transaction to handle RAW and WAW dependency. If a predecessor has been garbage collected, it is ignored (line 10). Otherwise, the node representing the predecessor is found or inserted if it does not exist already (line 11-16). The current node N is inserted to the predecessor's successors list and the *pred_size* is incremented accordingly. RAW and WAW dependencies are handle separately (line 17-23). Finally, *CheckRecoverability()* is called on the node to test whether it is recoverable and if so, whether it is ready for recovery right now (line 26).

The implementation of *CheckRecoverability()* is shown in Algorithm 5. Node N first checks whether all its RAW predecessors are recoverable (line 2). If so, N is recoverable and all its RAW successors should be notified (line 3-7). A whole subgraph of transactions may become recoverable as a consequence. The function also checks whether N is ready for recovery by checking if its *waw_pred_size* is 0. If so, the transaction is inserted to *recover_queue* and will be recovered by a worker thread (line 8-11).

Algorithm 4 Insert() method of the recovery dependency graph.

```
    # Active transactions are stored in a set called Nodes.
 1: function INSERT(R)
 2:     N ← Nodes.find(R)
 3:     if N = NULL then
 4:         N ← new Node(R)
 5:         N.state ← Unrecoverable
 6:         Nodes.insert(N)
 7:     end if
 8:     N.redo_info = R.redo_info
 9:     for all P in T.predecessors do
            # P can be ignored if it has been garbage collected
10:         if P.id ≥ P.logger.min_gc_id then
11:             N_P ← Nodes.find(P)
12:             if N_P = NULL then
13:                 N_P ← new Node(P)
14:                 N_P.state ← Unrecoverable
15:                 Nodes.insert(N_P)
16:             end if
17:             if P in R.raw_predecessors then
18:                 N.raw_pred_size ++
19:                 N_P.raw_successors.insert(N)
20:             else if P in R.waw_predecessors then
21:                 N.waw_pred_size ++
22:                 N_P.waw_successors.insert(N)
23:             end if
24:         end if
25:     end for
26:     CheckRecoverability(N)
27: end function
```

The garbage collection process removes nodes from the graph after they are recovered. However, removing a node immediately after recovery leads to incorrect behavior. Consider T1 and T2 in Figure 1 where T2 RAW depends on T1. During recovery, assume T1 is recovered and garbage collected before T2 is read from the log file. Now when T2 is inserted to the graph, the system does not know whether T1 has been garbage collected or has not been read from its log file yet. In the first case, we can ignore T1 while in the second case we have to insert a node for T1. Failing to detect the correct state of T1 causes incorrect recovery.

To resolve this uncertainty, we need an efficient way to tell whether a transaction has been garbage collected or not. We achieve this by garbage collecting transactions from the same log file in the transaction ID order. Since the ID of a transaction is its position in the log file, the transactions are garbage collected in the same order as they are read from the log file. The system only remembers a

Algorithm 5 Checking recoverability and enforcing recovery order.

```
 1: function CHECKRECOVERABILITY(N)
 2:     if N.raw_pred_size = 0 then
 3:         N.state ← Recoverable
 4:         for all S in N.raw_successors do
 5:             S.raw_pred_size −−
 6:             CheckRecoverability(S)
 7:         end for
 8:         if N.waw_pred_size = 0 then
 9:             N.state ← ReadyForRecovery
10:             recover_queue.push(N.record)
11:         end if
12:     end if
13: end function
```

Algorithm 6 Each logger garbage collects recovered nodes in the recovery dependency graph.

```
1: function GarbageCollect(R)
2:     gc_queue.enqueue(R)
3:     while gc_queue.head.state = Recovered do
4:         Nodes.delete(gc_queue.head)
5:         min_gc_id = gc_queue.head.id
6:         gc_queue.pop()
7:     end while
8: end function
```

minimal garbage collect ID (*min_gc_id*) for each log file, all and only transactions with smaller IDs have been garbage collected. This way, if T2 does not find T1 during insertion, it compares T1's ID with *min_gc_id* (line 10 in Algorithm 4). If T1's ID is smaller, it can be ignored, otherwise a new node is inserted for T1.

The garbage collection algorithm is shown in Algorithm 6. Each logger maintains a FIFO queue (*gc_queue*) of transactions that have been read from its corresponding log file but have not been garbage collected yet. Whenever the transaction at the head of the queue becomes *Recovered*, the node is deleted from the recovery dependency graph and the *min_gc_id* is updated (line 3-7). *min_gc_id* only monotonically increase since the log records are inserted to the *gc_queue* in order.

5. COMMAND LOGGING IN TAURUS

Command logging was proposed [18] to reduce the amount of REDO data in log records. Instead of logging the data modification made by a transaction, the database can log the command of the transaction instead. The command include the store procedure name of the transaction and the input parameters. Compared to data logging, command logging can reduce the size of logs by an order of magnitude, therefore significantly mitigate the I/O pressure.

Existing command logging algorithms, however, still use a single LSN to order all the transactions. Therefore, they cannot leverage multiple I/O drives unless the database is partitioned and a majority of transactions access a single partition. Furthermore, during a crash recovery, all the transactions from a log have to be sequentially executed, making the recovery time long.

In this section, we discuss how the dependency tracking in Taurus can increase parallelism in command logging and recovery. The parallel logging algorithm for data and command logging are identical. The only difference is that the REDO information contains commands instead of data. We will present the changes made to the parallel recovery algorithm.

5.1 RAW and WAR Dependency

In Section 4.4, we have proposed an algorithm of parallel recovery for data logging. However, the same algorithm does not work with command logging. To recover a command log record, the transaction needs to be re-executed. Therefore, it not only writes to the database, but also needs to read tuples from the database. Therefore, a transaction can start recovery only after all its WAW *and RAW* predecessors have been recovered. In contrast, recovery in data logging only requires WAW predecessors be recovered.

Furthermore, WAR dependencies also need special attention in command logging, as a transaction should not write to a tuple if some other transactions need to read the previous version of the tuple. One possible implementation is to enforce the recovery order for all the three types of dependencies (RAW, WAW, WAR). For most workloads, this should still expose large amount of parallelism during recovery. The downside, however, is that we need to explicitly track WAR dependencies.

One way of tracking WAR is to write a transaction's WAR predecessors to its log record. For each tuple a transaction modifies, it needs to find all the transactions reading the previous version of the tuple. This cannot be done as efficiently as tracking RAW and WAW, which only requires a single *last_writer* field. Alternative, WAR can also be inferred using RAW and WAW dependencies. If T2 RAW depends on T1 and T3 WAW depends on T1, then we may have T3 WAR depends on T2. This solution is more efficient than the first one, but still requires analysis over the whole graph. This makes it challenging to exploit the garbage collection idea discussed in Section 4.4.2.

5.2 Multi-version Parallel Recovery

In Taurus, we get around the tracking of WAR dependencies by maintaining multiple versions of each tuples during the recovery process. This way, transactions with WAR dependencies can be recovered in parallel. Since no tuples are overwritten, the reading transaction can always find the tuple it needs.

There are two ways to identify which version to read for a transaction. One approach is to use a timestamp-based concurrency control algorithm and include the commit timestamp in each transaction's log record. A transaction reads the tuple with the largest timestamp which is smaller than its commit timestamp. For concurrency control algorithm that do not have timestamps (e.g., Two-Phase Locking), the transaction can read the latest version whose last writer is in its RAW predecessor.

The recovery algorithm for command logging largely remains the same as for data logging discussed in Section 4.4.2. However, the recovery order has to following both RAW and WAW constraints. So a transaction is *ReadyForRecovery* when all its RAW and WAW predecessors are recovered.

We illustrate our algorithm, i.e. the multiversioning part, with the example from Figure 1. We consider the case where all the four transactions are found in log files.

The reconstructed graph will have look like Figure 1 without the WAR edge between T1 and T3. Then T1 and T3 can be recovered in parallel. T1 creates a new version of Y but T3 will read the previous version. T2 and T4 will be recovered after T1 and T3 are done, respectively.

5.3 Garbage Collection

The garbage collection in the recovery dependency graph is the same for both command and data logging. For command logging, however, we also need to garbage collect the stale versions of tuples to keep the memory footprint small. Therefore, the system needs to know when a version will never be accessed by a future transaction which may still stay in a log file right now.

Our solution to this problem is very similar to the garbage collection in a normal multiversion concurrency control algorithms. Let's first consider a timestamp-based solution. During normal execution, we may collect the minimal commit timestamp of active transactions and flush this min commit timestamp to log files periodically as sync timestamp. During recovery, if transactions from all log files have been recovered up to a particular sync timestamp, all the records with smaller timestamps (except the last version of each tuple) can be garbage collected.

The garbage collection algorithm can also be implemented without timestamps. In this case, threads should periodically synchronize to write a sync record to each log file, and promise all transactions after the synchronization should read the latest version of data before the synchronization. Namely, the sync records create a consistent cut of all the transactions in the system. After transactions from all

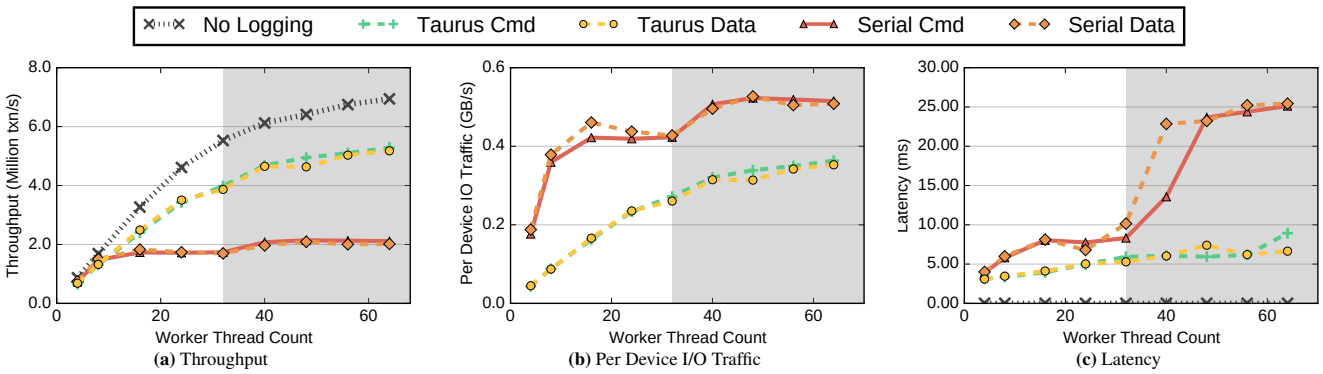


Figure 5: YCSB Logging Performance – Throughtput, latency, and I/O traffic of different logging algorithms with different number of worker threads.

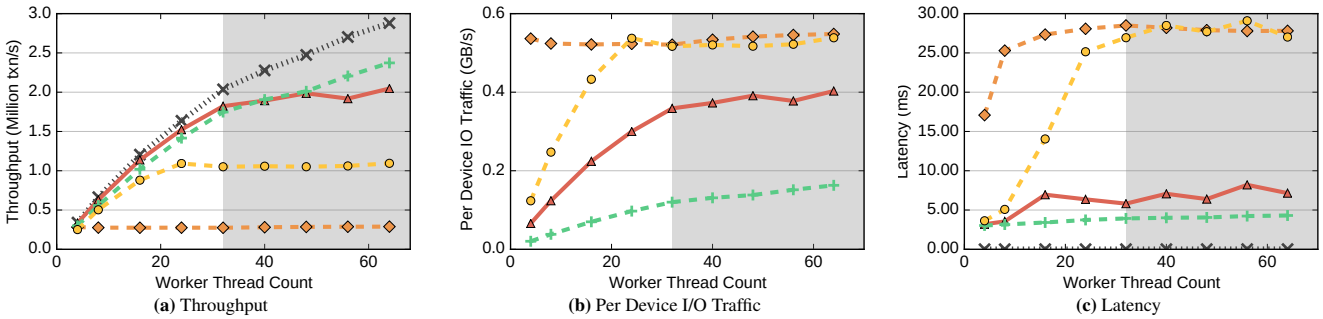


Figure 6: TPC-C Logging Performance – Throughtput, latency, and I/O traffic of different logging algorithsm.

log files are recovered to a particular sync record, all versions except the latest one can be garbage collected.

6. EXPERIMENTAL EVALUATION

We now provide an evaluation of Taurus. We implemented Taurus on the DBx1000 open-source DBMS [2]. DBx1000 uses the scalable timestamp ordering-based concurrency control algorithm [26].

We first present the experimental setup in Section 6.1 and workloads in Section 6.2. Then we compare the performance of Taurus to baseline data and command logging algorithms. Logging performance will be discussed in Section 6.3 and recovery performance in Section 6.4. We sweep the number of loggers in Section 6.5 and discuss the overhead of dependency tracking in Section 6.6.

6.1 Experimental Setup

All the experiments are executed on a server with four Intel Xeon E7-4830 processors with 32 physical cores in total. With hyperthreading turned on, the server supports 64 logical cores. The server contains three Fusion IO ioDrive2 flash drives [1] and one disk array with RAID-5. Together, they provide a combined I/O write throughput of about 2 GB/s. Taurus will write to all the devices through multiple loggers while a serial logging algorithm only writes to a single Fusion IO device.

We use group commit for each logger which maintains a in-memory buffer of size 16 MB. The buffer is flushed every 5 ms or if the buffer is half full, whichever happens first. Worker threads can write to the rest of the buffer when parts of it is being flushed. For each experiment, we execute 400 K transactions per thread for data logging and 1 M transactions per thread for command logging.

We compare the following logging schemes:

No Logging has no logging component. A transaction commits immediately after it pre-commits. This is the performance upper bound of any logging algorithm.

Serial Data is the ARIES [19] algorithm optimized for main memory DBMS. Each transaction creates a data log record using the after-images of tuples it modified. Transactions are logged to a single IO device. During recovery, all transactions are recoverd sequentially in the LSN order.

Serial Cmd is the algorithm proposed in [18]. Each transaction logs the command of the transaction (store procesure name and input parameters). During recovery, all the transactions are reexecuted sequentially in the LSN order.

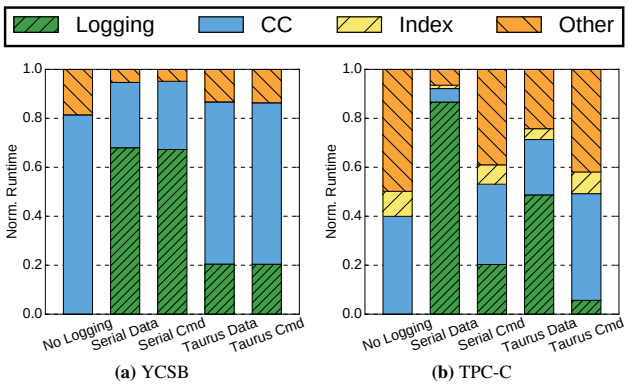
Taurus Data and **Taurus Cmd** are the two versions of Taurus presented in Section 4 and Section 5. They both log to multiple log files during normal execution.

6.2 Workloads

We evaluate logging performance using two workloads.

YCSB (Yahoo! Cloud Serving Benchmark) is representative of large-scale on-line services [8]. Each data tuple in YCSB has 10 fields, each of which is 100 bytes in size. In our evaluation, each transaction reads and writes to two random tuples and each query touches a single field of that tuple. All tuples are accessed with uniformly random distribution. This way we can eliminate the effect of other contention (e.g., concurrency control) and focus on the evaluation of the logging subsystem. Since the value written to a tuple is an input parameter, both data and command logging have the same log record size.

TPC-C is the industry standard for evaluating OLTP systems [22]. It consists of nine tables that simulate a warehouse-centric order processing application. We only model two (Payment and NewOrder) out of the five transactions in TPC-C since they make up the majority (88%) of the default TPC-C mix. Each transaction type comprises 50% of the transactions. Different from YCSB, the command log record of TPC-C is much smaller (10× on average) than the data log record. For each experiment, we populate the database with 32



(a) YCSB **(b)** TPC-C

Figure 7: Logging runtime breakdown at 64 threads.

warehouses with an initial size of 4 GB.

6.3 Logging Performance

Figure 5 shows the logging performance of YCSB in terms of throughput, latency and I/O traffic. The number of worker threads are swept on the x-axis. Ideally, a scalable algorithm sees an increase in its throughput as the thread count increases. For *No Logging*, this is achieved because there is no overhead involved in analyzing and writing log files. The performance of *Serial Data* and *Serial Command*, however, stops scaling after 2 million transactions per second. As shown in Figure 5b, they saturate the I/O bandwidth of a single Fusion IO card which is about 500 MB/s. *Taurus Data* and *Taurus Cmd*, in contrast, scale much better than serial algorithms by leveraging multiple I/O devices. Figure 5b shows that they never saturate the I/O bandwidth of all storage devices. Therefore, performance can further increase given more cores in the processor. After 32 cores, however, performance increasing slows down due to hyperthreading where two threads running on a single hardware core contend for hardware resources.

Figure 5c shows the latency of all the logging algorithms. The latency of *No Logging* stays less of 10 us and therefore is too small to be seen in this graph. Taurus algorithms have reasonably low latency which stays around 5 ms which is the group commit time interval. Serial algorithms have higher latency compared to Taurus due to I/O bandwidth saturation. When the thread count is large, the logging buffer quickly fills up which makes the flushing time to be more than 5 ms. This significantly increases the transaction latency.

Figure 6 shows the performance of TPC-C running on the same set of logging algorithms. Compared to our YCSB configuration, data logging in TPC-C has much larger log records and therefore generates more I/O traffic. As a result, *Serial Data* quickly saturates at 0.29 million transactions per second. *Taurus Data* can improve this performance by almost a factor of four (saturating at 1.09 million transactions per second). The performance difference can be better understood by looking at Figure 6b, where both *Serial Data* and *Taurus Data* saturate the bandwidth capacity of their I/O devices. *Taurus Data* performs better since it leverages four I/O devices but *Serial Data* only uses one I/O device.

Command logging on TPC-C has much better performance than data logging due to the significant reduction in logging traffic. Indeed, neither *Serial Cmd* nor *Taurus Cmd* saturate the I/O bandwidth. The performance of *Serial Cmd*, however, still stops increasing after 2 million transactions per second. Our scrutiny shows that the bottleneck is not because of the I/O traffic, but comes from the contention in the global LSN allocation and the shared in-memory buffer. The performance does not change even if we do not write to log files. *Taurus Cmd* removes this bottleneck by spreading it to four separate loggers. As a result, the per-device I/O traffic is also much low than the *Serial Cmd*. Note that at small thread counts where the contention on the logging subsystem is low, the performance of *Taurus Cmd* is slightly worse than *Serial Cmd* due to the cost of dependency tracking (i.e., last writer and predecessor vector). But the overhead is small in general.

Figure 6c shows the latency of all the logging configurations. Again, *No Logging* has an average latency less than 25 us and thus cannot be seen in the graph. At high thread count, both *Serial Data* and *Taurus Data* have high latency since they both saturate the I/O bandwidth. But *name Data* has lower latency at low thread count when the I/Os are not saturated yet. *Serial Cmd* and *Taurus Cmd* have much lower latency than the other two algorithms, and between them, *Taurus Cmd* has slightly lower latency as it has lower pressure on I/Os.

Figure 7 shows the logging time breakdown of different algorithms at 64 threads for both YCSB and TPC-C. The execution time is broken down to *Logging*, *CC* (Concurrency Control), *Index* and *Other* (which is the actual transaction logic). Taurus significantly reduces the logging time for both data and command logging as it spread the I/O contention across more devices.

6.4 Recovery Performance

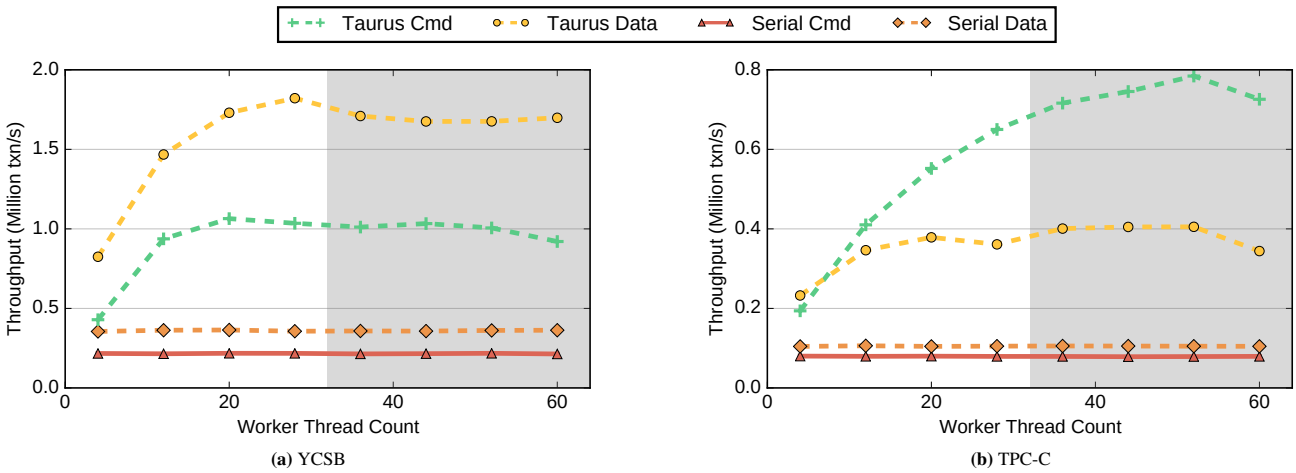
Figure 8 shows the recovery performance of different logging algorithms for YCSB and TPC-C when changing the number of worker threads. For *Serial Data* and *Serial Cmd*, a single I/O thread reads log records from the log file, and another worker thread recovers all the transactions in the LSN order. For these two algorithms, having more than one worker thread does not affect performance since the extra threads are not in use. The recovery throughput of *Serial Data* is slightly higher than *Serial Cmd*. This is because data recovery only requires copying the after images to the corresponding tuples while command recovery has to re-execute the logic of the transaction again, which is typically more expensive.

Both *Taurus Data* and *Taurus Cmd* have much higher throughput than the serial algorithms. *Taurus Data* is slightly better due to cheaper recovery cost. At high thread count, performance of both algorithms saturate. The bottleneck here are the I/O threads which need to read log records from the corresponding storage devices and update the dependency graph.

Figure 8 shows the recovery performance on TPC-C workload. The general trend is the same as with YCSB. However, *Taurus Cmd* outperforms *Taurus Data* when thread count is large. We found that *Taurus Data* almost saturates the I/O bandwidth with more than 32 threads. For *Taurus Cmd*, since the log records are much smaller, the I/O bandwidth does not saturate and thus performance scales better. At 64 threads, performance of both *Taurus Data* and *Taurus Cmd* drop slightly due to resource contention from hyperthreading.

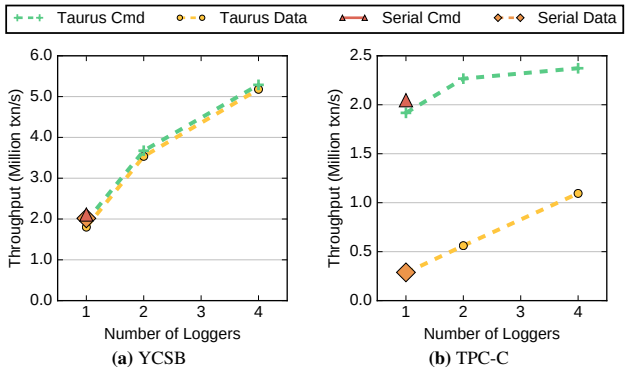
6.5 Scaling the number of loggers

Figure 9 shows the performance of Taurus as we increase the number of loggers for both YCSB (Figure 9a) and TPC-C (Figure 9b). We also show the performance of serial logging algorithms for reference. For both YCSB and TPC-C, when only a single logger exists, the logging performance of Taurus is almost the same as that of serial baselines. However, due to the overhead of dependency tracking in Taurus (discussed in more details in Section 6.6), the performance can be slightly worse. As the number of loggers increase, the performance of Taurus keeps increasing while serial baselines cannot leverage the extra I/O devices. We argue that serial logging would be a good fit for machines with a single persistent storage device but Taurus would perform better for machines with more than one such devices. Taurus can also significantly speed up the recovery process by leveraging parallelism.



(a) YCSB **(b)** TPC-C

Figure 8: Recovery throughput using YCSB and TPC-C benchmarks.



(a) YCSB **(b)** TPC-C

Figure 9: Logging performance with different number of loggers at 64 threads.

6.6 Dependency Tracking Overhead

In this section, we discuss the overheads of dependency tracking in Taurus. We discuss this in terms of logging overhead and recovery overhead.

During the logging process, the extra metadata requirement was discussed in Section 4.1. Specifically, each tuple requires a *last writer* field and we need a commit dependency table to track predecessor vectors for uncommitted transactions. The required in-memory space of these two structures are not significant. Some extra computation is required to manage these structures but the overhead is small in general as shown in Figure 5 and Figure 6.

One source of overhead in Taurus is the dependency information (e.g., predecessor list) in each log record, which was not required for serial logging algorithms. Table 1 shows the average log record size breakdown of both *Taurus Data* and *Taurus Cmd* on YCSB and TPC-C. For YCSB, the REDO information has almost the same size for data and command logging. The dependency information of command logging is higher since it includes the commit timestamp of the transaction which is used for garbage collection (cf. Section 5.3). Overall, the dependency information only adds about 10% storage overhead.

For TPC-C, *Taurus Data* logs more REDO information than YCSB and the dependency information only incurs a small overhead. For *Taurus Cmd*, however, the REDO information is very small and as a result the dependency information comprises a big portion of log data. This does not kill Taurus's performance since the log data can distribute to multiple devices. For each device, the amount of log data is still less than that of *Serial Cmd*.

Bytes/Record	Taurus Data		Taurus Cmd	
	REDO	Dep. Info.	REDO	Dep. Info.
YCSB	252.0	20.5	244.0	30.6
TPC-C	1899.7	68.5	196.9	77.8

Table 1: Average log record size (in Bytes) for Taurus.

It is possible to compress the predecessor list of a transaction by sacrificing the available parallelism during recovery. For example, instead of precisely tracking each individual predecessor, we can just track the predecessors with the maximal address in each log file. So for predecessors in the same file, only one of them is tracked. During recovery, however, we can no longer construct the complete dependency graph. A transaction can start recovery not only after its predecessors, but also after all transactions from the same log file before its predecessors. For certain workloads, the recovery process may still show abundant parallelism even after predecessor compression. For these workloads, it is worthwhile to trade some recovery parallelism for storage efficiency.

7. RELATED WORK

ARIES [19] has been the gold standard in database logging and has been widely implemented. The scalability of logging in multi-core processors has been studied recently by some related work [13, 23, 24, 27]. Aether [13] optimized the implementation of ARIES by removing lock related bottlenecks, but still uses a singe logger. [27] and [24] allow the logging algorithm to take advantage of multiple persistent devices by logging transactions in batches. Each batch can be flushed to multiple devices in parallel. They also removed the global LSN allocation bottleneck. However, both algorithms only work for data logging.

Pure logical logging and recovery were first proposed in [17] which logs the table IDs and keys of modified tuples instead of page IDs. The algorithm, however, still uses data logging. Command logging was first proposed in [18] which pushes logical logging to extremes. Command logging has been applied to systems like Hyper [15], H-Store [14] and its commercial successor VoltDB [3]. Current command logging algorithms are implemented on partitioned databases where different partitions can log and recover in parallel. Taurus enables parallel command logging for unpartitioned databases.

Adaptive logging [25] proposed parallel recovery for command logging based on the dependency graph in a distributed partitioned database. Different from Taurus, it does not directly maintain the dependency information, but rather infers it from the transactions'

read/write set. Adaptive logging also requires each transaction to log their start and end time which requires clocks to be synchronized across the system.

Fast crash recovery based on NVM (Non-Volatile Memory) has been a active research area recently [4, 5, 11, 7, 16]. This line of work leverages the high read/write throughput and byte-addressable nature of NVM to accelerate the logging and recovery performance. The dependency tracking technique in Taurus can be applied to NVM based systems to leverage the large number of loggers for parallel logging and recovery.

Early lock release (ELR) [9, 21] allows a transaction to release locks before flushing to log files. [9] also uses dependency tracking to paralell log flushing. But unlike Taurus, it does not log dependency information to the log records. This leads to two shortcomings. First, dependent transactions have to be flushed to persistent storage in order. This is hard to achieve when the dependency graph is complex [13]. Second, it is hard to recovery transactions in parallel.

8. CONCLUSION

This paper proposed a new logging algorithm Taurus that breaks the sequential abstraction in traditional data and command logging algorithms. By tracking and logging fine-grained dependency information among transactions, Taurus allows log records to be written to multiple persistent devices in arbitrary order. It also speeds up the crash recovery by recovering independent transactions in parallel. Taurus shows the good potential of fine-grained dependency tracking in building fault tolerant systems.

9. REFERENCES

[1] Data sheet fusion-io® iodrive®2 solid-state storage devices. `https://sp.ts.fujitsu.com/dmsp/Publications/public/ds-py-pci-ssd-ioDrive2.pdf`.

[2] DBx1000. `https://github.com/yxymit/DBx1000`.

[3] VoltDB. `http://voltdb.com`.

[4] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722. ACM, 2015.

[5] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *Proc. VLDB Endow.*, 2016.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. 1987.

[7] A. Chatzistergiou, M. Cintra, and S. D. Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment*, 8(5):497–508, 2015.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC'10*, pages 143–154.

[9] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[10] 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.

[11] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1221–1231. IEEE, 2011.

[12] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[13] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *Proc. VLDB Endow.*, 3(1-2):681–692, 2010.

[14] 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.

[15] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE, 2011.

[16] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706. ACM, 2015.

[17] D. Lomet, K. Tzoumas, and M. Zwilling. Implementing performance competitive logical recovery. *Proceedings of the VLDB Endowment*, 4(7):430–439, 2011.

[18] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615. IEEE, 2014.

[19] 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. *TODS*, 17(1):94–162, 1992.

[20] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE transactions on Software Engineering*, (3):240–247, 1983.

[21] E. Soisalon-Soininen and T. Ylönen. Partial strictness in two-phase locking. In *International Conference on Database Theory*, pages 139–147. Springer, 1995.

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

[23] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[24] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, 2014.

[25] 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 *Proceedings of the 2016 International Conference on Management of Data*, pages 1119–1134. ACM, 2016.

[26] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of SIGMOD 2016*, 2016.

[27] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 465–477. USENIX Association, 2014.