Rethinking Main Memory OLTP Recovery

Nirmesh Malviya #1, Ariel Weisberg *2, Samuel Madden #3, Michael Stonebraker #4

#MIT CSAIL *VoltDB Inc.

[1]nirmesh@csail.mit.edu [2]aweisberg@voltdb.com [3]madden@csail.mit.edu [4]stonebraker@csail.mit.edu

***Abstract*—Fine-grained, record-oriented write-ahead logging, as exemplified by systems like ARIES, has been the gold standard for relational database recovery. In this paper, we show that in modern high-throughput transaction processing systems, this is no longer the optimal way to recover a database system. In particular, as transaction throughputs get higher, ARIES-style logging starts to represent a non-trivial fraction of the overall transaction execution time.**

We propose a lighter weight, coarse-grained *command logging* technique which only records the transactions that were executed on the database. It then does recovery by starting from a transactionally consistent checkpoint and replaying the commands in the log as if they were new transactions. By avoiding the overhead of fine-grained logging of before and after images (both CPU complexity as well as substantial associated I/O), command logging can yield significantly higher throughput at run-time. Recovery times for command logging are higher compared to an ARIES-style physiological logging approach, but with the advent of high-availability techniques that can mask the outage of a recovering node, recovery speeds have become secondary in importance to run-time performance for most applications.

We evaluated our approach on an implementation of TPC-C in a main memory database system (VoltDB), and found that command logging can offer 1.5× higher throughput than a main-memory optimized implementation of ARIES-style physiological logging.

I. INTRODUCTION

Database systems typically rely on a recovery subsystem to ensure the durability of committed transactions. If the database crashes while some transactions are in-flight, a recovery phase ensures that updates made by transactions that committed pre-crash are reflected in the database after recovery, and that updates performed by uncommitted transactions are not reflected in the database state.

The gold standard for recovery is write-ahead logging during transaction execution, with crash recovery using a combination of logical UNDO and physical REDO, exemplified by systems like ARIES [20]. In a conventional logging system like ARIES, before a modification to a database page is propagated to disk, a log entry containing an image of the modified data in the page before and after the operation is logged. Additionally, the system ensures that the tail of the log is on disk before a commit returns. This makes it possible to provide the durability guarantees described above.

There is an alternative to ARIES-style logging, however. Suppose during transaction execution, instead of logging modifications, the transaction's logic (such as SQL query statements) is written to the log. For transactional applications that run the same query templates over and over, it may in fact be possible to simply log a transaction identifier (e.g., a stored procedure's name) along with the query parameters; doing so also keeps the log entries small. Such a *command log* captures updates performed on the database implicitly in the commands (or transactions) themselves, with only one log record entry per command. After a crash, if we can bring up the database using a pre-crash transactionally-consistent snapshot (which may or may not reflect all of the committed transactions from before the crash), the database can recover by simply re-executing the transactions stored in the command log in serial order instead of replaying individual writes as in ARIES-style physiological logging.

Compared to physiological logging, command logging operates at a much coarser granularity, and this leads to important performance differences between the two approaches. Generally, command logging will write substantially fewer bytes per transaction than physiological logging, which needs to write the data affected by each update. Command logging simply logs the incoming transaction text or name, while physiological logging needs to spend many CPU cycles to construct before and after images of pages, which may require differencing with the existing pages in order to keep log records compact. These differences mean that physiological logging will impose a significant run-time overhead in a high throughput transaction processing (OLTP) system. For example, as shown by [7], a typical transaction processing system (Shore) spends 10-20% of the time executing TPC-C transactions (at only a few hundred transactions per second) on ARIES-style physiological logging. As transaction processing systems become faster, and more memory resident, this will start to represent an increasingly larger fraction of the total query processing time. For example, in high throughput data processing systems like H-Store [28], RAMCloud [21] and Redis [27], the goal is to process many thousands of transactions per second per node. To achieve such performance, it is important that logging be done efficiently. Up to this point, it has been an open question as to whether disk-based logging can even be used in such a system without sacrificing throughput. In this paper we show definitively that it is quite feasible!

It is also relevant to look at recovery times for the two logging approaches. One would expect physiological logging to perform recovery faster than command logging; because in command logging, transactions need to be re-executed completely at recovery time whereas in ARIES-style physiological logging, only data updates need to be re-applied. However, given that failures are infrequent (once a week or less), recovery times are generally much less important than run-time

performance. Additionally, any production OLTP deployment will likely employ some form of high-availability (e.g., based on replication) that will mask single-node failures. Thus, failures that require recovery to ensure system availability are much less frequent.

In this paper, our goal is to study these performance trade-offs between physiological logging and command logging in detail. We describe how command logging works, and discuss our implementation of both command logging and a main-memory optimized version of physiological logging in the VoltDB main memory open-source database system [30] (VoltDB is based on the design of H-Store [28]). We compare the performance of both the logging modes on two transactional benchmarks, Voter and TPC-C.

Our experimental results show that command logging has a much lower run-time overhead compared to physiological logging when (a) the transactions are not very complex and only a small fraction of all transactions are distributed, so that CPU cycles spent constructing the differential physiological log record and the disk I/O due to physiological logging represent a substantial fraction of transaction execution time; and (b) the size of a command log record written for a transaction is small and the transaction updates a large number of data tuples, because physiological logging does much more work in this case. In our experiments, we found that for TPC-C, which has has short transactions that update a moderate number of records, the maximum overall throughput achieved by our system when command logging is used is about 1.5× higher than the throughput when physiological logging is employed instead, a result in line with the plot's prediction. Also for TPC-C, we found that recovery times, as expected, are better for physiological logging than command logging, by a factor of about 1.5.

Given this high level overview, the rest of this paper is organized as follows. We begin with a short discussion of VoltDB's system architecture in Section II. We then describe our approach to command logging in Section III, followed by a detailed description of our main-memory adaptation of ARIES-style physiological logging in Section IV. Subsequently, we report extensive performance experiments in Section V and discuss possible approaches to generalize command logging in Section VI. Section VII provides an overview of relevant past work in this area, and Section VIII summarizes the paper.

II. VOLTDB OVERVIEW

VoltDB is an open source main memory database system whose design is based on that of the H-Store system [28], with some differences. Below, we give a brief overview of VoltDB's system architecture.

A. Partitions and Execution Sites

VoltDB is a distributed in-memory database which runs on a cluster of nodes. In VoltDB, a table is horizontally partitioned on keys; each partition resides in the main memory of a cluster node and can be replicated across several nodes for high availability. All indexes are also kept in main memory along with the partition. Every node in the cluster runs multiple execution *sites* (e.g., one per CPU core), with each partition on the node assigned to one such site. Each node has an *initiator* component which sends out transactions to the appropriate partitions/replicas. By employing careful, workload-aware partitioning, most transactions can be made *single-sited* (run on just a single partition) [24].

B. Transactions

Transactions in VoltDB are issued as *stored procedures* that run inside of the database system. Rather than sending SQL commands at run-time, applications register a set of SQL-based procedures (the workload) with the database system, with each transaction being a single stored procedure. This scheme requires all transactions to be known in advance, but for OLTP applications that back websites and other online systems, such an assumption is reasonable. Encapsulating all transaction logic in a single stored procedure prevents application stalls mid-transaction and also allows VoltDB to avoid the overhead of transaction parsing at run-time. At run-time, client applications invoke these stored procedures, passing in just the procedure names and parameters.

All transactions in VoltDB are run *serially* at the appropriate execution site(s). Because OLTP transactions are short, typically touch only a small number of database records, and do not experience application or disk stalls, this is actually more efficient than using locking or other concurrency control mechanisms which themselves introduce significant overhead [11].

VoltDB supports distributed and replicated transactions by running all transactions in a globally agreed upon order. Since only one transaction can run at a time at each execution site, a transaction that involves all partitions will be isolated from all other transactions. If a transaction operates at a single partition, it is isolated from other transactions because the execution site owning the partition is single threaded and all replicas of the partition/site run transactions in the globally agreed upon order. Even if one or more sites do not respond to a transaction request (e.g., because of a crash), the transaction will be executed as long as a one replica of each partition involved in the transaction is available.

Below, we briefly explain how transaction ordering works in VoltDB.

1) Global Transaction Ordering and Replication: In VoltDB, a database component called an *initiator* receives client requests and dispatches transactions to the appropriate execution sites; the pool of initiators consists of one initiator for each node. Each initiator generates unique timestamp-based transaction-ids that are roughly synchronized with those generated by other initiators using NTP. At each execution site, transactions received from an initiator are placed in a special priority queue, which ensures that only tasks that are globally ordered and safely replicated are executed.

For global ordering, this is done by checking if the id of a transaction in the queue is the minimum across prior transactions received from all initiators. Since initiators uses

timestamps to generate monotonically increasing transaction-ids and messages to the priority queue are TCP ordered, the minimum transaction-id can be used to determine when the position of a transaction in the global order is known. If a transaction is not globally ordered, it is held in the queue until its position in the global order is known.

Replication follows a similar process, but in reverse: initiators inform the transaction execution sites of the minimum safely replicated transaction-id for transactions from that initiator. If a transaction-id is greater than the safely replicated transaction-id for that initiator, the site holds the transaction in the queue until it is replicated.

This global ordering and replication information propagates via new transaction requests and their responses and not as separate messages. In the event of light transaction load, heartbeats (no-op transactions) are used to prevent stalls.

We note that a recently released version of VoltDB does ordering differently, above we have described how global transaction ordering works in the system we have used to implement our recovery approaches.

C. Durability

The VoltDB mechanisms discussed above result in very high transaction throughputs (about 4K transactions per second per core on TPC-C), but durability is still a problem. In the event of a single node failure, replicas ensure availability of database contents. Additionally, VoltDB uses command logging (described in Section III), along with a non-blocking transaction-consistent checkpointing mechanism to avoid loss of database contents in the event of power failure or other cluster-wide outage.

To deal with scenarios where a transaction must be rolled back mid-execution, VoltDB maintains an in-memory undo log of compensating actions for a transaction. Thus, transaction *savepoints* (partial rollback) are also supported; this is possible because any partial rollback for a deterministic transaction will also be deterministic. The undo log is separate from the command log and is never written to disk. It is discarded on transaction commit/abort because it can be regenerated when the transaction is replayed.

1) Asynchronous Checkpointing: VoltDB's checkpoint mechanism periodically writes all committed database state to disk (index updates are not propagated to disk). Before starting the snapshot, a distributed transaction is used to start the snapshot by putting all of the sites into a *copy-on-write* (COW) mode. The snapshot process then begins scanning every row in the table, while queries continue to execute on the database. All updates from this point until the end of the snapshot are COW if they are performed on a row that hasn't been scanned for the snapshot yet. Specifically, three bits per row track whether the row was added, deleted, or modified since the snapshot began (these bits are not a part of the snapshot). Newly added rows are skipped by the snapshot. Deleted rows are removed by the snapshot after they have been scanned. Updates cause the row to be copied to a shadow table before the update is applied, so the snapshot can read the shadow version (as with deletes, the shadow version is removed after the checkpoint process scans it). A background process serializes the snapshot to disk, so there is no need to quiesce the system and the checkpoint can be written asynchronously. When one such sweep is done, and the background process has completed its scan, the executor returns to regular mode.

This checkpointing mechanism, although somewhat VoltDB specific, can be easily generalized to any database system that uses snapshots for isolation, since the copy-on-write mode is very similar to the way transaction isolation is implemented in snapshot-isolation based systems.

Having a transaction-consistent checkpoint is crucial for the correctness of the command logging recovery protocol, as we shall discuss in the next section which details how our implementation of command logging in VoltDB works.

III. Command Logging

The idea behind command logging is to simply log what *command* was issued to the database before the command (a transaction for example) is actually executed. Command logging is thus a write-ahead logging approach meant to persist database actions and allow a node to recover from a crash. Note that command logging is an extreme form of *logical logging*, and is distinct from both physical logging and record-level logical logging. As noted in Section I, the advantage of command logging is that it is extremely lightweight, requiring just a single log record to be written per transaction.

For the rest of this paper, we assume that each command is a stored procedure and that the terms *command logging* and *transaction logging* are equivalent. The commands written to the log record in a command logging approach thus consist of the name of a stored procedure and the parameters to be passed to the procedure. For stored procedures that must generate random numbers or call non-deterministic functions such as `date()`/`time()`, a timestamp based transaction-id (see Section II) can be used as the seed for the generator and for extracting the date/time.

Generally speaking, stored procedure names are likely to be substantially smaller than entire SQL-queries, so this serves to reduce the amount of data logged by command logging. Specifically, an entry in the log is of the form `(transaction-name, parameter-values)`.

A. Writing to the Log

Writing a command log record for a single-partition transaction is relatively simple. For a distributed transaction, only the coordinator site specific to the transaction writes the transaction to its command log; all other sites participating in the transaction *do not* log the transaction. The coordinator for a distributed transaction is the site with the lowest id on the node where the transaction was initiated. Multiple execution sites on the same node write to a shared command log. For both single and multi-sited transactions, if replicas are present, the transaction is also logged at all replicas of the site. The

check-sum	LSN	record-type	xaction-id	partition-id	xaction-type	params

Fig. 1. Command logging record structure.

command log for each node also records transaction ordering messages sent/received by the node's initiator at runtime.

Transactions are written to the command log right away after they have been received, thus a transaction need not have been globally ordered and replicated before it's written to the log. This in turn requires that at recovery time, transactions be sorted again in a manner that agrees with the global transaction ordering at runtime. This is easy to accomplish because the logged ordering messages are also replayed at recovery time.

In our VoltDB implementation of command logging, log records written out for each transaction have the structure shown in Figure 1.

1) Optimizations: Command log records can be flushed to disk either synchronously or asynchronously. ACID semantics can be guaranteed only with synchronous logging, because in this case a log record for a transaction is forced to disk before the transaction is acknowledged as committed. For this reason, even though our implementation permits either mode, we report results only for the synchronous mode and throughout the rest of this paper, use the term *command logging* to mean synchronous command logging.

To improve the performance of command logging, we employ group-commit: the system batches log records for multiple transactions (more than a fixed threshold or few milliseconds worth) and flushes them to the disk together. After the disk write has successfully completed, a commit confirmation is sent for all transactions in the batch. This batching of writes to the command log reduces the number of writes to the disk and helps improve synchronous command logging performance, at the cost of a small amount of additional latency per-transaction.

B. Recovery

Recovery processing for the command logging approach works as follows.

First, using the latest database snapshot on disk, database contents are initialized in memory. Because the disk snapshot does not contain indexes, all indexes are then rebuilt at start-up; this can be done in parallel with the snapshot restore as index reconstruction for the part of the database that has already been restored can begin while the rest of the database finishes loading.

Next, the shared command log for each node is read by a dedicated thread which reads the log into memory in chunks. Starting from the log record for the first transaction not reflected in the database, log entries are processed by the node's initiator and the corresponding transaction is dispatched to the appropriate sites (which may be on a different node in case of a distributed transaction).

This recovery approach works even if the number of execution sites at replay time is different from the number of sites at run-time, as long as the number of database partitions remains the same. In the event of a site topology change, the initiator replaying the log can simply send the transaction to the new site for a given partition-id.

Given that each log record corresponds to a single transaction and that the initiator has access to ordering messages written to the command log at run-time, global ordering at replay time is identical to the pre-crash execution ordering.

If a command log record for a transaction is written to the log but the database system crashes before the transaction completes executing (so that the client isn't notified), command log replay will recover this transaction and bring the database to a state as if this transaction had been committed, even though the client won't be notified after replay (a similar situation can happen in a conventional DBMS as well).

We further discuss how command logging could be extended to other database systems in Section VI.

IV. Physiological Logging

Traditional database systems have typically employed ARIES [20] or ARIES-like techniques for their recovery subsystem. ARIES does physiological logging; each operation (insert/delete/update) performed by a transaction is written to a *log record table* before the update is actually performed on the data. Each such log entry contains the before and after images of modified data. Recovery using ARIES happens in several passes, which include an analysis pass, a physical REDO pass and a logical UNDO pass.

While the core idea behind ARIES can be used in a main-memory database, substantial changes are required for the technique to work in a main memory context. In addition, the main-memory environment can be exploited to make logging more efficient. Given the differences, throughout the rest of this paper, we simply refer to our main memory optimized ARIES-style logging technique as *physiological logging*.

Below, we discuss in detail the changes required and optimizations that must be made for main-memory physiological logging to work well.

A. Supporting Main Memory

In a disk-based database, inserts, updates and deletes to tables are reflected on disk as updates to the appropriate disk page(s) storing the data. For each modified page, ARIES writes a separate log record with a unique *logical sequence number* (LSN) (a write to a page is assumed to be atomic [26]). These log records contain disk specific fields such as the page-id of the modified page along with length and offset of change. This is stored as a RID, or record ID, of the form `(page #, slot #)`. A *dirty page table*, capturing the earliest log record that modified a dirty page in the buffer pool is also maintained. In addition, a *transaction table* keeps track of the state of active transactions, including the LSN of the last log record written out by each transaction. The dirty page and transaction tables are written out to disk along with periodic checkpoints.

Checksum	LSN	Record-type	Insert/Update/ Delete	Transaction-id	Partition-id	Table Name	Primary Key	Modified Column List	Before Image	After Image

Fig. 2. Physiological logging record structure.

In a main-memory database like VoltDB, a data tuple can be accessed directly by probing its main-memory location without any indirection through a page-oriented buffer pool. Thus, the ARIES logging structures can be simplified when adapted to main-memory physiological logging; specifically, all disk related fields in the log record structure can be omitted.

For each modification to a database tuple, our physiological logging approach simply writes a unique entry to the log with serialized before and after images of the tuple. Instead of referencing a tuple through a `(page #, slot #)` RID, the tuple is referenced via a `(table-id, primary-key)` pair that uniquely identifies the modified data tuple. If a table does not have a unique primary key and the modification operation is not an insert, the entire before-image of a tuple must be used to identify the tuple's location in the table either via a sequential scan or a non-unique index lookup. For the both the Voter and TPC-C benchmarks we use in our study, all tables written to have primary keys except the TPC-C History table which only has tuples inserted into it (see Section V-A for schema details).

Use of persistent virtual memory addresses instead of `(table-id, primary-key)` for in-memory tuple identity is also an option [13][8], but we believe that it is not a good choice as a unique identifier because a database in general is not guaranteed to load a table and all its records at the same virtual address on restart after a crash unless specifically engineered to do so. Moreover, doing so limits potential compaction over the table memory layout to minimize data fragmentation.

1) Optimizations: In this section we describe a number of optimizations to our physiological logging scheme.

Differential Logging. For tables with wide rows, a large amount of log space can be saved by additionally recording which attributes in the tuple were updated by a transaction, with before and after images recorded for only those columns instead of the entire tuple (this optimization does not apply to inserts). The logging scheme is thus *physiological* – physical with respect to the changes for a particular column and logical with respect to which columns have been modified (similar to the way ARIES records physical changes to logical page slots). We found that that this *differential logging* optimization led to a significant reduction in a log record's size for the TPC-C benchmark (nearly $5\times$ for TPC-C). However, we noticed that this reduction came at the cost of increased CPU complexity to construct log records, an overhead which becomes significant at in-memory OLTP execution throughputs (we discuss the implications of this observation in Section V).

Dirty Page Tracking. An in-memory database has no concept of disk pages and so unlike ARIES, we do not need to maintain a dirty page table. One option is to create a *dirty record table* to keep track of all dirty (updated or deleted) database records. For a write-heavy workload, though regular snapshotting would keep the size of this table bounded, it can grow to a fairly large size. Alternatively, we could eliminate the separate dirty record table and instead simply associate a dirty bit with each database tuple in memory. This dirty bit is subsequently unset when the dirty record is written out to disk as a part of a snapshot. Not storing the dirty record table results in space savings, but depending on the checkpoint mechanism in use, doing so can have significant performance impacts, as we discuss next.

Checkpointing. Disk-based ARIES assumes fuzzy checkpointing [18] to be the database snapshot mechanism. Fuzzy checkpoints happen concurrently with regular transaction activity, and thus updates made by uncommitted transactions can also be written to disk as a part of the checkpoint. In disk-based ARIES, both the dirty page and transaction tables are flushed to disk along with each checkpoint. The main memory equivalent of this would be to write out the dirty record and transaction tables with a checkpoint. Not having an explicit dirty record table in such a scenario is inefficient, because prior to each checkpoint, we would need to scan the in-memory database to construct the *dirty record table* so it could be written along with the checkpoint.

Alternatively, we could use transaction-consistent checkpointing [25] instead of fuzzy checkpointing. VoltDB already uses non-blocking transaction-consistent checkpointing (see Section II), so we leveraged it for our implementation. With transaction consistent checkpointing, only updates from committed transactions are made persistent, so that we can simply keep track of the oldest LSN whose updates have not yet been reflected on disk. Thus, the dirty record table is not needed at checkpoint time.

Moreover, as explained in Section II-C1, transaction-consistent checkpointing is also used by our system to ensure correctness of command logging.

Log-per-node. In our VoltDB implementation of physiological logging, execution sites on the same node write to a shared log with arbitrary interleaving of log records from different sites. The ordering of log records per site is still preserved. A field in the log record identifies the partition the update corresponds to (site-id to partition-id mapping is one-to-many as each site can be host to more than one database partition). Having a shared log for all sites as opposed to a log per-execution site makes recovery much simpler, since the database is not constrained to restarting with an identical partition-to-site mapping on a given node. This is important, because if a node crash requires a reconfiguration, or the database must be recovered on a different machine from the one it was previously running on, we may have a different number of sites and a different partition-to-site mapping. However, the shared nature of the log requires that all partitions previously residing together on a node must still be on the same node for replay, even though the number of sites on the node can be changed.

Batched writes. Because OLTP transactions are short, the amount of log data produced per update in the transaction

is not enough to justify an early disk write given that the final update's log record must also be flushed before the transaction can commit. For this reason, its best to buffer all log records for a single transaction and write them all to the log together.

Similar to ARIES, our physiological logging is synchronous, so that log writes of a committed transaction are forced to disk before we report back the transaction's status as committed. Similar to command logging, our physiological logging implementation uses group commit; writes from different transactions are batched together to achieve better disk throughput and to reduce the logging overhead per transaction.

The log record structure for our physiological logging implementation in VoltDB is shown in Figure 2.

B. Recovery

Recovery using disk-based ARIES happens in three phases: an analysis phase, a redo phase and an undo phase. The redo pass is physical and the undo pass is *logical*.

Our physiological logging scheme for main-memory also begins with an analysis phase, the goal of which is to identify the LSN from which log replay should start. The redo pass then reads every log entry starting from this LSN and reapplies updates in the order the log entries appear. For each log entry, the data tuple that needs to be updated is identified using the `(table-name, primary-key)` pair and the serialized after-image bytes in the log record are used to modify the tuple appropriately (this covers insert and delete operations as well). For the primary-key lookup identifying a tuple's location to be fast, an index on the primary key is used at replay time. In VoltDB, index modifications are not logged to disk at run-time, so all indexes are reconstructed at recovery time in parallel with snapshot reloading prior to log replay (see Section III).

Because log records corresponding to different partitions of the database can be replayed in any order, the redo phase is highly parallelizable. This optimization yielded linear scale up in recovery speeds with the number of cores used for replay (see Section V for performance numbers). Next comes the undo pass. For transactions which had not committed at the time of the crash, the undo phase simply scans the log in reverse order using the transaction table and uses the before image of the data record to undo the update (or deletes the record in case it was an insert).

Recovery can be simplified for an in-memory database such as VoltDB that uses transaction consistent checkpointing and only permits serial execution of transactions over each database partition. In such a system, no uncommitted writes will be present on disk. Also, because transactions are executed in serial order by the run-time system, log records for a single transaction writing to some partition on an execution site S are never interleaved with log records for other transactions executed by S. Hence for each partition, only the transaction executing at the time of crash will need to be rolled back (at most one per partition). Even this single rollback can be avoided by simply not replaying the tail of the log corresponding to this transaction; doing so necessitates a one transaction look-ahead per partition at replay time. Then during crash recovery, no rollbacks are required and the undo pass can be eliminated altogether (employing this optimization reduced log record sizes by nearly a factor of two, as the before image in update records could now be omitted). Also, with no undo pass, the transaction table can be done away with.

Note that in databases other than VoltDB which use transaction-consistent checkpointing but run multiple concurrent transactions per execution site, the idea of simply not reapplying the last transaction's updates for each site does not work and an undo pass is required. This is because there could be a mixture of operations from committed and uncommitted transactions in the log.

V. PERFORMANCE EVALUATION

We implemented both command logging and ARIES-style physiological logging inside VoltDB, with group-commit enabled for both techniques. The logging is synchronous in each case with both techniques issuing an `fsync` to ensure that a transaction's log records are written to disk before the results are returned. All performance optimizations discussed in Sections III and IV were implemented and enabled for all experiments except where we explicitly study the performance impact of turning off a specific optimization. We implemented an additional optimization for physiological logging, in which all the log records for each transaction are first logged to a local buffer, and then at commit time, written to the disk in a batch along with records of other already completed transactions. For OLTP workloads, this optimization adds a small amount of latency but amortizes the cost of synchronous log writes and substantially improves throughput. Also, we ensured that the physiological logging implementation group-commits with the same frequency as command logging.

In this section, we show experimental results comparing command logging against physiological logging. We study several different performance dimensions to characterize the circumstances under which one approach is preferable over the other: *run-time overhead* (throughput and latency), *recovery time* and *bytes logged per transaction.*

We also look at the effect of distributed transactions and replication on performance of the two techniques.

In Section V-A, we briefly discuss the benchmarks we used in our study. Then we describe our experimental setup in Section V-B followed by performance results in Section V-C. Finally, we summarize our results and discuss their high level implications in Section V-D.

A. Benchmarks

We use two different OLTP benchmarks in our study, Voter[1] and TPC-C. These two benchmarks differ considerably in their transaction complexity. The work done by each transaction in the Voter benchmark is minimal compared to TPC-C. TPC-C database tables are also much wider and exhibit more complex relationships as compared to Voter.

The two benchmarks are described below.

[1] https://community.voltdb.com/node/47

1) Voter: The Voter benchmark simulates a phone based election process. The database schema is extremely simple and is as follows:

```
contestants (contestant_name STRING,
             contestant_number INTEGER)
area_code_state (areacode INTEGER, state STRING)
votes (vote_id INTEGER, phone_number INTEGER,
      state STRING, contestant_number INTEGER)
```

Given a fixed set of contestants, each voter can cast multiple votes up to a set maximum. During a run of the benchmark, votes from valid telephone numbers randomly generated by the client are cast and reflected in the `votes` table. At the end of the run, the contestant with the maximum number of votes is declared the winner.

This benchmark is open-loop and has only one kind of transaction, the stored procedure `vote`. This transaction inserts one row into the `votes` table and commits. There are no reads to the `votes` table until the end of the client's run, the other two tables in the database are read as a part of the vote transaction but not written to. In addition, the width of all the tables is very small (less than 20 bytes each).

The number of contestants as well as the number of votes each voter is allowed to cast can be varied. For our experiments, these are set to default values of 6 and 2 respectively.

2) TPC-C: TPC-C [29] is a standard OLTP system benchmark simulating an order-entry environment.

The TPC-C database consists of nine different tables: Customer, District, History, Item, New-Order, Order, Order-Line, Stock and Warehouse. These tables are between 3 and 21 columns wide and are related to each other via foreign key relationships. The Item table is read-only.

The benchmark is a mix of five concurrent transactions of varying complexity, namely New-Order, Payment, Delivery, Order-Status and Stock-Level. Of these, Order-Status and Stock-Level are read-only and do not update the contents of the database. The number of New-Order transactions executed per minute (tpmC) is the metric used to measure system throughput.

The TPC-C implementation used to report numbers in Section V-C differs from the standard benchmark in that (a) it's open-loop, (b) New-order transactions do not read items from a remote warehouse, so that the transactions are always single-sited. Performance numbers for multi-sited TPC-C New-order transactions however are reported in Section V-C6.

As TPC-C simulates a real order-entry environment, the benchmark description also mimics a human terminal operator by adding keying times and think times for each transaction. Our implementation of TPC-C does not take these into account.

B. Experimental setup

All our experiments in Sections V-C1-V-C5 were run on a single Intel Xeon dual-socket 2.4 GHz 8-core server with 24GB of RAM, 12TB of hard disk with a battery backed write cache and running Ubuntu Server. To improve disk throughput, the disk was mounted with appropriate additional flags while ensuring that data durability was not compromised; such a careful setup is necessary to optimize either recovery system.

The client was run on a separate machine with a system configuration identical to that of the server. We simulated several clients requesting transactions from the server by running a single client issuing requests asynchronously at a fixed rate.

For our distributed transactions experiments in Section V-C6, we used a cluster of four identical machines, with one machine used to run an asynchronous client and the other three used as database servers. Each machine was an Intel Xeon dual-socket 12-core server box with a processor speed of 2.4GHz, 48GB of RAM, 4TB of hard disk space with a battery backed cache and running Ubuntu Server.

Because the VoltDB server process runs on a multi-core machine, we can partition the database and run several execution sites concurrently, with each site accessing its own partition. For an 8-core machine, we experimentally determined that running six sites works best for the Voter benchmark and more sites did not lead to increased throughput. For the TPC-C benchmark, we found that best performance is achieved by using all possible sites (one per core). Each site corresponds to one warehouse, so that the results to follow are for a TPC-C 8-warehouse configuration (except for Section V-C6, where 12 warehouses are used). While it's possible to fit a much larger database (e.g., 64 warehouses) given the server memory, we found that the system throughput for a 64-warehouse configuration was nearly the same as for the 8-warehouse one (which is expected given that the entire database is in memory in both cases). Given that the TPC-C database grows in size over time as new transactions are issued, we chose a smaller database to facilitate long running experiments without running out of memory.

C. Results

All the experimental results we present below were obtained by running our benchmarks against three different modes of VoltDB: (a) command logging on and physiological logging turned off, (b) physiological logging turned on and command logging turned off, and (c) both command logging and physiological logging turned off.

For most direct performance comparisons between the above three modes, we show plots with the performance metric on the y-axis and the client rate on the x-axis. Doing so allows us to compare performance of the three logging modes by asking a simple question: what throughput/latency/recovery-rate do each of the logging modes have for a given amount of work to be done per unit time (in this case a client attempting to execute transactions at a certain rate)?

For all experiments, we set the system snapshot frequency to 180 seconds. Increasing or lowering this value affects performance of each logging mode equally as the system does extra work in the background at runtime in all cases. The rationale for setting the snapshotting frequency to the order of a few minutes instead of seconds (or continuous) is that there is substantial data on the log that must be replayed, which

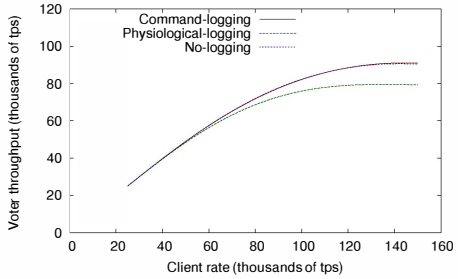

Fig. 3. Voter throughput vs. client rate (both tps).

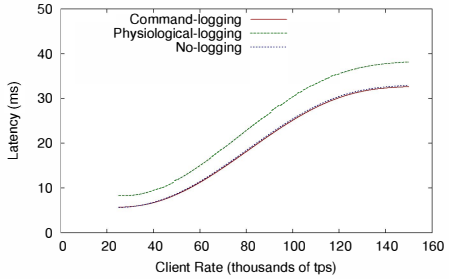

Fig. 4. Voter latency in milliseconds vs. client rate (tps).

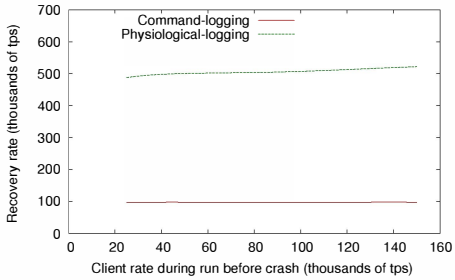

Fig. 5. Voter log replay rates (tps).

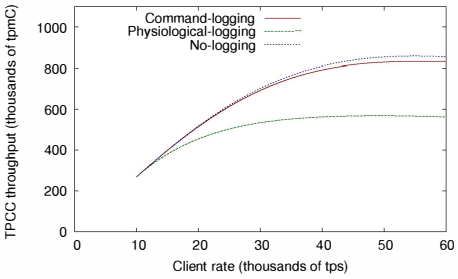

Fig. 6. TPC-C throughput (tpmC) vs. client rate (tps).

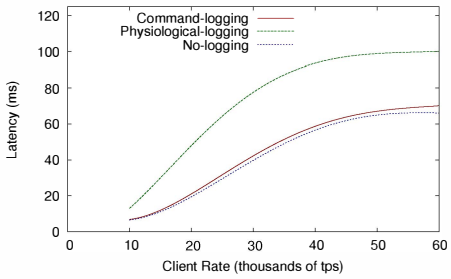

Fig. 7. TPC-C latency in milliseconds vs. client rate (tps).

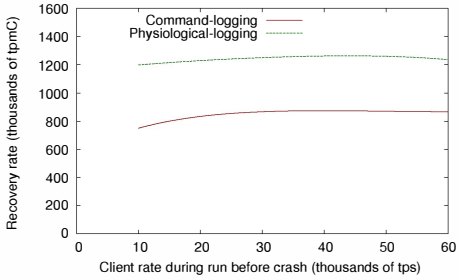

Fig. 8. TPC-C log replay rates (tpmC).

makes measured recovery rates more reliable and offsets any dominating replay startup costs that would affect the measured numbers.

1) Throughput: Figure 3 shows the variation in system throughput for the voter benchmark as the client rate is varied from 25,000 transactions per second up to 150,000 transactions per second. All three logging modes (no-logging, physiological-logging and command-logging) are able to match the client rate until 80K tps at which physiological logging tops out while the other two saturate at 95K tps. We observe that the overhead of command logging is nearly zero. Due to the extra CPU overhead of creating a log record based on the insert row's serialized bytes during the transaction, physiological logging suffers about 15% drop in maximum throughput at run time. For more complex transactions, physiological logging has a higher performance penalty, as we see next.

Figure 6 shows throughput measured in tpmC achieved by the three logging modes for the TPC-C benchmark, as the client rate varies from 10K up to 60K tps. Similar to the results for the voter benchmark, command logging achieves nearly the same throughput as the no logging scenario. However, here physiological logging caps out at about 66% of the throughput achieved by the other two. In other words, command logging provides about 1.5× more throughput than physiological logging for the TPC-C benchmark. This is expected behavior because TPC-C transactions are much more complex than voter transactions, and each one potentially updates many database records. Extra CPU overhead is incurred in constructing log record for each of these inserts/updates, and the amount of logged data also increases (see Section V-C3 for numbers). The penalty on Voter is lower because the number of log writes for the `vote` transaction is small (just one).

Both approaches have short transactions, do better with command logging, but TPC-C performs more updates per transaction, and is favored more heavily by command logging.

2) Latency: The variation of transaction latency with client rates for the voter benchmark is shown in Figure 4. For client rates less than 50K tps, the system runs well under its capacity and all logging methods result in a 5-7ms latency. Note that this latency is dependent on the group commit frequency, which was fixed at 5ms for this experiment (obtained by varying group commit frequencies is an independent experiment, elided due to space constraints). The latencies for all methods gradually increase as the database server approaches saturation load. Command-logging has almost the same latency as no-logging whereas physiological-logging has a 15% higher latency. The higher transaction latencies for client rates greater than the saturation load result from each transaction waiting in a queue before it can execute. The queue itself only allows a maximum of 5,000 outstanding transactions, and the admission control mechanism in VoltDB refuses to accept new transactions if the queue is full.

In Figure 7, we see that TPC-C performs similarly, except that physiological logging reaches saturation at about 21K tps, so that its latency goes up much earlier. The other two logging modes hit saturation latencies at client rates higher than 30K tps and both have about the same latency. Due to extra logging overhead, physiological logging suffers from latencies that are at least 45% higher for all client rates.

3) Number of Bytes Logged: As noted earlier, the voter benchmark only has one transaction (the stored procedure *vote*). For each transaction initiated by the client, command logging writes a log record containing the name of this stored procedure and necessary parameters (phone number and state) along with a log header. We found that the size of this log record is always 55 bytes. On the other hand, physiological logging directly records a new after-image (insert to the *votes* table) to the log along with a header, and writes 81 bytes per invocation of `vote`. This transaction only inserts data, so

that the before-image does not exist. Moreover, as discussed in Section IV, before images can be done away with in any case. For voter, both the logging techniques only write one log record per transaction.

The TPC-C benchmark has three different transaction types which update the database: *delivery*, *neworder* and *payment*. The above mentioned three different transaction types for TPC-C together modify 8 out of 9 tables in the TPC-C database (the *item* table is read-only). Modifications include insert, update as well as delete operations on tables. In many cases, only 1 record is modified per transaction for each table, but the *neworder*, *orders*, *order-line* and *stock* tables have either 10 or 100 records modified per transaction for certain operations.

For command logging, the three transactions write between 50 (*delivery*) and 170 (*neworder*) bytes per transaction (there is only one log record per transaction). The *neworder* transaction logs the highest number of bytes, which is not surprising given that *neworder* is the backbone of the TPC-C workload. Depending on the table that is updated, log record sizes for physiological logging vary from 70 bytes (New-Order table) to 240 bytes (Customer table) per record, with most log records less than 115 bytes in size. Overall, for TPC-C, physiological logging writes about 10× more data per transaction in comparison to command logging (averaged over the three transaction types).

4) Log Record Size vs Performance: Because physiological logging writes so much more data than command logging on TPC-C, we wanted to test if the run-time performance difference between the two systems on this benchmark was completely attributable to I/O time. We ran an experiment in which we truncated the size of physiological logging records written out per transaction to 100 bytes, which is approximately what command logging writes on an average for a TPC-C transaction. The resulting recovery log is unrecoverable/corrupt, but this is not important for the purposes of this experiment. We found that physiological logging throughput slightly increases by a mere 1%, and command logging wins by nearly the same factor.

Thus, the performance gap at run-time between command logging and physiological logging is a result of not only the extra disk I/O that physiological logging needs to do to write larger records to disk, but also of the higher CPU overhead incurred in logging activities during transaction execution. As discussed in Section IV, this overhead incurred by physiological logging is due to CPU cycles spent generating I/O efficient differential log records. While the CPU complexity of creating log records is not a new phenomenon, it becomes significant at main-memory OLTP speeds, where the actual work performed by each transaction is small and completes in tens to hundreds of microseconds.

5) Recovery Times: After a server node crashes and is brought up again, it must recover to its initial state by first reading the latest database snapshot into memory with indexes rebuilt in parallel and then replaying log records. For both voter and TPC-C, snapshot restore and index reconstruction take the same amount of time irrespective of the logging mode being used. If no logging was done at run-time, all transactions executed after the last snapshot was written to disk will be permanently lost. Hence, our recovery performance numbers are for command logging and physiological logging only. Our implementations for both the logging modes are optimized to do parallel log replay, each execution site reads from the shared recovery log and replays all log records corresponding to its site.

Figure 5 shows the log replay times for the two logging modes for voter. During recovery, the system replays the log at maximum speed but does not serve new client transactions simultaneously, naturally this way recovery rate is not a function of previous load. Command logging must actually re-execute each transaction, and we see that its 100K tps recovery rate is about the same as the maximum throughput it can achieve at run-time (seen earlier in Figure 3). On the other hand, for voter, physiological logging is able to replay the log almost 5× faster at about 500K tps. This difference is due to the fact that physiological logging directly records each transaction's modifications to the log at run-time. It does not have to repeat its reads or transaction logic during recovery and is able to recover much faster. The simplicity of voter transactions ensures that the physiological logging overhead of parsing each log record and reapplying the relevant updates is small.

In Figure 8, we see that even for the TPC-C benchmark, physiological logging also replays at a faster rate compared to command logging. Command logging can only recover at about 865K tpmC, which is also its maximum run-time throughput on an average (Figure 6). However owing to the increased complexity of TPC-C transactions, physiological logging replay is only about 1.5× faster than command logging for TPC-C as opposed to the 5× speedup for the much simpler voter benchmark.

While command logging has a longer recovery time, it's impact on availability is minimal because all modern production OLTP systems are engineered to employ replication for high availability, so that the higher throughput of command logging at run-time is a good tradeoff for it's slower background recovery while the other replica nodes continue to serve live traffic.

Recovery numbers in the two plots just discussed are for log replay only and do not include log read times. Once a database snapshot has been restored from disk, the log is read in chunks by a single execution site and thereafter shared by all sites on the node during replay; this applies for both command logging and physiological logging. For both Voter and TPC-C, the log read in case of command logging added less than 1% extra overhead to the replay time, due to small log records and relatively high per-transaction replay times. Log reads in physiological logging, in contrast, add a 30% overhead to voter replay times and about 8% overhead to TPC-C, due to larger log records and faster re-execution times.

6) Distributed Transactions and Replicated Partitions: As we noted in Section I, OLTP transactions have become shorter

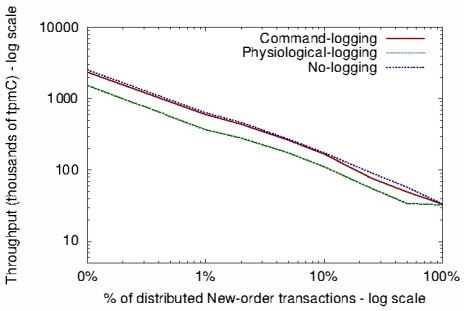

Fig. 9. TPC-C New-order run-time throughput (tpmC) vs. % of distributed transactions for a single-node multi-site setup.

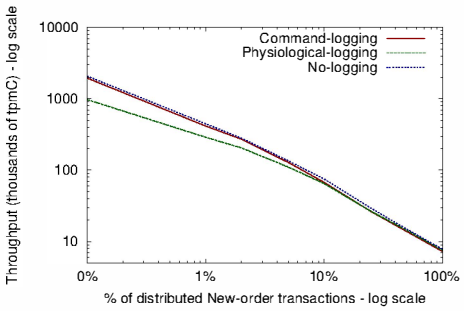

Fig. 10. TPC-C New-order run-time throughput (tpmC) vs. % of distributed transactions for a multi-node 3-way replicated multi-site setup.

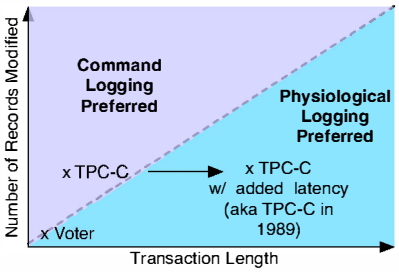

Fig. 11. Illustration of when command logging is preferred over write-ahead physiological logging, with experimental results overlaid.

as processors have gotten faster and RAM sizes of tens of gigabytes have become routine. In this section, we increase the average transaction length by varying the fraction of distributed transactions in the workload and see how run-time performance of each of the three logging approaches changes as we do so. Our hypothesis is that a longer transaction length should make physiological logging look better, because logging will represent a small fraction of the total work the transaction does.

For all our experiments in this section, we use a modified TPC-C benchmark consisting of 100% New-Order transactions and vary the fraction of multi-partition New-Order transactions. The methodology behind doing so is as follows. In TPC-C New-Order, an order has between 5 to 15 items, for an average of 10. Each item can come from a remote warehouse with x% probability (default is 1%, we vary this). Our TPC-C New-order table is partitioned on warehouse-id, so for a New-order transaction to be multi-partition, at least one of the items must come from a remote warehouse, and thus the probability that a transaction is distributed can be approximated as $1 - \left(1 - \frac{x}{100}\right)^{10}$.

As mentioned in Section V-B, we use a slightly different cluster setup for running our distributed transactions/replication experiments. As servers we use for these experiments have 12 cores each, we employ a configuration with 12 execution sites per server node. Our New-order table is partitioned on warehouse id, and we let each warehouse partition be owned by an execution site, so that New-order transactions with an item from a remote warehouse are always multi-partition.

We start with results for a system setup similar to results presented previously: an asynchronous client issuing transactions at maximum speed to a single server node with multiple execution sites with no replication. Figure 9 shows throughput numbers for this case as we increase the number of multi-partition transactions (latency plots are omitted due to space constraints, but latency is inversely related to throughput as shown in previous results.) Because throughput drops dramatically with even a small fraction of distributed transactions, both axes on the plot are in log scale with the 0% x-label (only single-sited transactions) approximated as 0.1% on the plot. Here the numbers for no distributed transactions ($x = 0$) are in agreement with those seen in earlier sections, with command logging having a throughput of 2.4M tpmC (slightly below that of no-logging at 2.6M tpmC) and 1.5× that of physiological logging 1.5 tpmC throughput. This 1.5× throughput gap between command logging and physiological logging remains even as distributed transactions are introduced. This gap slowly drops down, and remains about 1.4× even at 50% distributed transactions, until at about 100% distributed transactions, transaction latencies are so high that all logging approaches provide identical results.

Figure 10 shows performance numbers for the different logging approaches when we have a cluster configuration of 3 server nodes running 12 execution sites each, with each site replicated three-ways. We still have a TPC-C workload with 12 warehouses, with the difference that now each warehouse partition is stored by three different execution sites. We see that again, at 0%, the performance gap is as expected, command logging throughput wins by a factor of almost 2×, with a penalty of less than 5% compared to when no logging is done. For this configuration however, the performance offered by all three approaches drops quickly as we increase the fraction of distributed transactions, with a gap of 1.2× in favor of command logging at 5% distributed transactions, which closes down to nearly identical throughput for all three approaches beyond 10%. These results are in agreement with the hypothesis: progressively higher transaction lengths lead to smaller run-time performance gaps between the different logging approaches.

Another interesting point to note is that the gap between the different logging approaches closes slower with no replication and faster with replication (Figures 9 vs. 10) : this is expected because a 3-way replication setup makes each transaction in the workload, distributed or not, multi-sited.

We do not show recovery rates due to lack of space here, but we found that physiological logging is much more efficient at recovery compared to command logging if the workload has a very high fraction of distributed transactions.

D. Discussion

Our results shows that command logging has a much lower run-time overhead than physiological logging (nearly zero in fact). This is due to the fact that it does less work at run-time to generate log records, and also because it writes less data to disk. In the two benchmarks we evaluated, command logging was able to achieve as much as a 1.5× performance improvement over our main-memory optimized

implementation of physiological logging on TPC-C, and about 1.2× on Voter. This improved performance comes at the cost of an increased recovery time for command logging, since it has to redo all of the work of a transaction, whereas physiological logging only has to re-apply updates to data tuples. Recovery times for command logging range from 1.5× slower on TPC-C to 5× slower on Voter. In reality, system failures are infrequent, and can be masked via high-availability through replication; this makes recovery speed secondary in importance to system performance for most systems. Hence, in modern high-throughput settings, command logging, with its near-zero overhead at run-time and modest reduction in recovery times, is the best choice.

In our experiments with high fraction of distributed transactions, physiological logging does better, since the overheads represent a small fraction of overall run-time, and recovery times for physiological logging become *much* better than for command logging. Hence, for applications with complex or mostly distributed transactions that update few records (which is not true of most OLTP applications), ARIES-style physiological logging is probably a better choice. This is also the reason why ARIES has traditionally been considered the gold-standard method of recovery: in the 1980's when initial research on recovery was done, OLTP throughputs were much lower, and the relative overheads of ARIES-style logging likely represented a much smaller fraction of the total work done per transaction. These results are summarized Figure 11.

Our conclusion is that for modern OLTP database systems that need to process many thousands of transactions per second, command logging should be the recovery method of choice, unless for some reason, recovery times are unusually important for the OLTP system.

VI. Generalizing Command Logging

A natural question about the command-logging approach described in this paper is how it would generalize to traditional disk-based systems and to other main-memory OLTP systems that use locking. We believe it should generalize well. To make it work, we need to ensure two properties: first, command log-based recovery needs to start from a transactionally-consistent snapshot, and second, replaying transactions in the command log in serial order must result in a re-execution that is equivalent to the original execution order of the committed transactions pre-crash.

To ensure the first property, if transactions are short-lived, there should be no need to write dirty (uncommitted) data to disk. However, this alone isn't sufficient to ensure that the state of the database on disk when recovery begins is transactionally consistent, since a crash may occur while data is being flushed back, resulting in only part of a transaction's state being on disk at recovery time. We may be able to atomically flush a set of pages to disk by relying on batteries in enterprise class disks to ensure that a set of flushed writes actually make it to disk even in the event of a power outage or crash. Alternatively, the same transactionally-consistent snapshotting approach used in VoltDB could be employed in a disk-based database by issuing a read-only transaction that reads the entire database and writes its pages to disk. If the database employs some form of snapshot-isolation (which most databases, including Postgres, Oracle, and SQL Server do), such read-only transactions will not block any other transactions in the system. However, this requires two copies of the database to be on disk, which may not be feasible. Exploring the best method for transactionally-consistent snapshotting of conventional databases, such as those in [25], is an interesting area for future work.

For the second property, assuming a transactionally-consistent checkpoint is available, serial replay from a command log will result in a correct recovery as long as the transactions in the log represent the serial equivalent commit order in which transactions were executed pre-crash. This will be the case assuming: (a) the use of strict two-phase locking (S2PL) for isolation, (b) no writes of dirty pages of uncommitted transactions, obviating the need for undo logging, so that correctness is ensured despite potential non-deterministic transaction aborts resulting from deadlocks. Other transactional isolation protocols, like serializable snapshot isolation (SSI) [1], unfortunately do not guarantee that commit order is the same as the serial equivalent execution order. Furthermore, it's unclear what the semantics of command log-based recovery are in the face of non-serializable isolation levels like snapshot isolation (which is widely used in practice). Hence, another interesting area for future work involves investigating this relationship.

VII. Related Work

ARIES [20] is considered the gold standard method for recovery in traditional databases.Most main memory database recovery techniques proposed in the past [8][9][4][15] are similar in spirit to ARIES; we briefly go over them here, a detailed discussion can be found in [5][6].

Dewitt et al [3] suggest compressing the log size by writing only new values to disk but require the presence of stable memory large enough to hold the write-ahead log for active transactions. In absence of such storage, they flush log records in batches (*group commit*). Both logging modes in our system (command logging and physiological logging) implement the group commit optimization.

Li et al [17] also suggest run-time optimizations for reducing log size by using shadow pages for updates but also require all shadow updates as well as the log buffer to reside in non-volatile memory. Lehman and Carey's recovery algorithm [14] also requires presence of non-volatile RAM to be able to store log tails. We do not make such an assumption in our system, which is impractical on commodity machines, the entire main memory contents are considered lost after a crash.

Levy and Silberschatz [16] describe an incremental recovery algorithm for main memory databases, which does not require recovery to be performed in a quiescent state, allowing transaction processing in parallel. This is achieved by recovering database pages individually. VoltDB does not have a concept of pages; we implement a similar idea by employing parallel recovery at a per partition level for physiological logging.

Achieving the same is harder with command logging owing to uncertainty about what pages a stored procedure would touch.

Purely logical logging has also been proposed recently [19]. Our work in this paper applies the logical logging idea in its extreme to an in-memory database similar in spirit to [12], and quantifies via extensive experiments the trade-offs between a highly logical command logging vs. a more traditional ARIES-style physiological logging approach.

Recent work by Cao et al describes main-memory checkpoint recovery algorithms for frequently consistent applicationsl [2], we believe their efficient checkpointing techniques can be used in combination with our recovery algorithms for better system performance.

Related work such as [10][22][23] has focused on making logging more efficient in general by employing ideas such as reducing log related lock contention. They emphasize that a separation of transactions from detailed knowledge about data placement naturally requires logical recovery. Our system architecture does not employ locking, so these techniques do not apply.

VIII. CONCLUSION

In this paper, we compared the run-time and recovery performance of command logging to ARIES-style physiological logging in high-throughput OLTP settings. Command logging recovers by re-running committed transactions from a transactionally-consistent checkpoint, whereas for physiological logging, fine-grained updates are recorded at run-time and the same updates applied at recovery time. We implemented these techniques in the VoltDB main-memory database system and found that on a modern machine running two OLTP benchmarks at high throughputs (in excess of 4K tps per core), physiological logging imposes significantly higher run-time overheads than command logging, yielding 1.2× to 1.5× lower throughput. It does, however, recover more quickly, with recovery times ranging from 1.5× to 5× faster. Our conclusion from these experiments is that, since most systems invoke recovery infrequently, databases focused on high-throughput transaction processing should implement command logging as the recovery system of choice.

We believe that these results should also apply to disk-resident databases, since logging represents a significant overhead in these systems as well (hundreds of microseconds per transaction, according to prior research [7]). Hence, generalizing command logging to a disk-based system is an interesting area of future work. Doing so is non-trivial as our current implementation of command logging relies on the fact that our system recovers from a transactionally-consistent checkpoint (which does not include any uncommitted data) and that the command log is written in an equivalent serial order of execution of the committed transactions in the database.

REFERENCES

[1] M. J. Cahill, U. Röhm, and A. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, 2009.

[2] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. SIGMOD '11, pages 265–276. ACM, 2011.

[3] 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. SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM.

[4] M. H. Eich. Main memory database recovery. In *Proceedings of 1986 ACM Fall joint computer conference*, ACM '86, pages 1226–1232, 1986.

[5] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4:509–516, 1992.

[6] L. Gruenwald, J. Huang, M. H. Dunham, J.-L. Lin, and A. C. Peltier. Recovery in main memory databases, 1996.

[7] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM.

[8] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dalí: A high performance main memory storage manager. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB'94*, pages 48–59.

[9] H. V. Jagadish, A. Silberschatz, and S. Sudarshan. Recovering from main-memory lapses. VLDB '93, pages 391–404.

[10] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *Proc. VLDB Endow.*, 3:681–692, September 2010.

[11] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. SIGMOD '10, pages 603–614, New York, NY, USA, 2010. ACM.

[12] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society.

[13] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Commun. ACM*, 34(10):50–63, Oct. 1991.

[14] T. J. Lehman and M. J. Carey. A recovery algorithm for a high-performance memory-resident database system. SIGMOD '87, pages 104–117, New York, NY, USA, 1987. ACM.

[15] T. J. Lehman and M. J. Carey. A concurrency control algorithm for memory-resident database systems. FOFO '89, pages 490–504, London, UK, 1989. Springer-Verlag.

[16] E. Levy and A. Silberschatz. Incremental recovery in main memory database systems. *IEEE Trans. on Knowl. and Data Eng.*, 4:529–540, December 1992.

[17] X. Li and M. H. Eich. Post-crash log processing for fuzzy checkpointing main memory databases. In *ICDE*, pages 117–124, Washington, DC, USA, 1993. IEEE Computer Society.

[18] J.-L. Lin and M. H. Dunham. Segmented fuzzy checkpointing for main memory databases. SAC '96, pages 158–165, New York, NY, USA, 1996. ACM.

[19] D. Lomet, K. Tzoumas, and M. Zwilling. Implementing performance competitive logical recovery. *Proc. VLDB Endow.*, 4:430–439, April 2011.

[20] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17:94–162, March 1992.

[21] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. In *SIGOPS OSR*. Stanford InfoLab, 2009.

[22] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3:928–939, September 2010.

[23] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. Plp: page latch-free shared-everything oltp. *Proc. VLDB Endow.*, 4:610–621, July 2011.

[24] A. Pavlo, C. Curino, and Z. Stan. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. SIGMOD '12, 2012.

[25] S. Pilarski and T. Kameda. Checkpointing for distributed databases: Starting from the basics. *IEEE Trans. Parallel Distrib. Syst.*, 3(5):602–610, Sept. 1992.

[26] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[27] Redis. http://redis.io.

[28] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07*, pages 1150–1160. VLDB Endowment, 2007.

[29] The TPC-C benchmark. www.tpc.org/tpcc.

[30] VoltDB. http://voltdb.com.