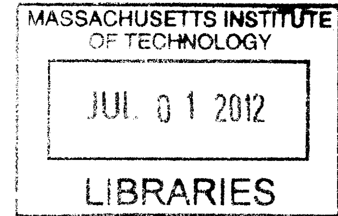
Recovery Algorithms for In-memory OLTP Databases


ARCHIVES
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 01 2012
LIBRARIES

by

Nirmesh Malviya

B.Tech., Computer Science and Engineering, Indian Institute of Technology Kanpur (2010)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

© Massachusetts Institute of Technology 2012. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 18, 2012

Certified by ....
Samuel Madden
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by ....
Michael Stonebraker
Adjunct Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Leslie A. Kolodziejski
Chair, EECS Committee on Graduate Students

Recovery Algorithms for In-memory OLTP Databases

by

Nirmesh Malviya

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

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 thesis, 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, page-level logging of before and after images (and substantial associated I/O), command logging can yield significantly higher throughput at run-time. Recovery times for command logging are higher compared to ARIES, but especially 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.

Thesis Supervisor: Samuel Madden
Title: Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Michael Stonebraker
Title: Adjunct Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my advisors, Sam Madden and Mike Stonebraker, for guiding me throughout the course of this work. This thesis would not have been possible without the mentoring and encouragement I received from both of them throughout the past academic year.

Sam's willingness to discuss research problems in detail just about any time I needed his help has been a source of great support to me ever since I started graduate school at MIT. There is much I have learned from him about research: he has been greatly instrumental in helping me improve my presentation and writing skills.

Mike's subtle insights into database research problems and his ability to keep the big picture in mind at the same time have constantly amazed me. I particularly thank him for suggesting that I work on this problem for my thesis.

I also thank my many friends in CSAIL – particularly Alvin, Eugene, Adam, Evan, Carlo, Arvind and Lenin for all the great help and advice.

Finally, I thank my parents and family for their love and unwavering belief in me and my abilities.

Contents

1 Introduction **13**

1.1 Command Logging: An Alternative Approach to Logging and Recovery 14

1.1.1 Comparison with ARIES 14

1.2 Why the Data Durability Problem Needs to be Revisited 15

1.3 Thesis Contributions 17

1.3.1 High-level Overview 17

1.4 Road Map 18

2 VoltDB Overview **19**

2.1 Architecture 19

2.2 Transactions as Stored Procedures 19

2.3 Concurrency Control and Durability 20

2.3.1 Checkpoint Mechanism 20

3 Command Logging **23**

3.1 Logging Transactions as Stored Procedures 23

3.1.1 Writing a Log-record 24

3.1.2 Optimizing for Performance 24

3.1.3 Log-record Structure 25

3.1.4 Transaction Rollbacks 25

3.2 Recovery 26

3.2.1 Transaction Replay 26

3.2.2 Correctness 26

4 **ARIES for Main Memory** **27**
4.1 Supporting Main Memory 27
4.1.1 Identifying a Modified Tuple 28
4.1.2 Dirty Record Table 29
4.1.3 Other Optimizations 30
4.1.4 Log-record Structure 30
4.2 Recovery for Main-memory 31
4.2.1 REDO Phase 31
4.2.2 UNDO Phase 31

5 **Performance Evaluation** **33**
5.1 Benchmarks 34
5.1.1 Voter 34
5.1.2 TPC-C 35
5.2 Experimental Setup 35
5.3 Results 36
5.3.1 Throughput 36
5.3.2 Latency 38
5.3.3 Number of Bytes Logged 39
5.3.4 Log Record Size vs. Performance 43
5.3.5 Recovery Times 44
5.3.6 Group Commit Frequency 46
5.3.7 Transaction Length 48
5.4 Discussion 52

6 **Generalizing Command Logging** **55**
6.1 Transactionally-consistent Snapshotting 55
6.2 Equivalent Transaction Replay Order 56

7 **Related Work** **57**

8 **Conclusion** **61**

List of Figures

1-1 Illustration of when command logging is preferred over ARIES. 16

3-1 Command logging record structure. 25

4-1 ARIES log record structure. 31

5-1 Voter benchmark schema. 34

5-2 Voter throughput vs. client rate (both tps). 37

5-3 TPC-C throughput (tpmC) vs. client rate (tps). 37

5-4 Voter latency in milliseconds vs. client rate (tps). 39

5-5 TPC-C latency in milliseconds vs. client rate (tps). 40

5-6 Number of data tuples modified per transaction for different TPC-C tables. Note that the y-axis is on a log scale. 41

5-7 Number of bytes written per ARIES log record for different TPC-C tables. 41

5-8 Number of bytes written per log record by command logging for different stored procedures. 42

5-9 Logging mode's TPC-C throughput does not merely depend on the number of log bytes written to disk. 43

5-10 Voter log replay rates (tps). 45

5-11 TPC-C log replay rates (tpmC). 46

5-12 Voter latency (ms) vs group commit frequency (ms). 47

5-13 TPC-C latency (ms) vs group commit frequency (ms). 48

5-14 Voter run-time throughput (tps) vs. added transaction length (ms). . 49

5-15 TPC-C run-time throughput (tpmC) vs. added transaction length (ms). 50
5-16 Voter replay rates (tps) on log scale vs. added transaction length. . . 51
5-17 TPC-C replay rates (tpmC) on log scale vs. added transaction length (ms). 52
5-18 Illustration of when command logging is preferred over write-ahead logging, with experimental results overlaid. 53

List of Tables

Chapter 1

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 [26]. In a conventional logging system like ARIES, before each modification of a page in the database is propagated to disk, a log entry containing an image of the state of 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.

Below we propose an alternative to the well accepted ARIES logging and recovery technique.

1.1 Command Logging: An Alternative Approach to Logging and Recovery

Suppose during transaction execution, instead of logging modifications, the transaction's logic (such as SQL query statements) is written to the log instead. 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 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.

1.1.1 Comparison with ARIES

Compared to ARIES, command logging operates at a much coarser granularity. This leads to important performance differences between the two. Generally, command logging will write substantially fewer bytes per transaction than ARIES, which needs to write the data affected by each update. Command logging simply logs the incoming transaction text or name, while ARIES needs to construct before and after images of pages, which may require differencing with the existing pages in order to keep log records compact.

Run-time

The differences mentioned above mean that ARIES will impose a significant run-time overhead in a high throughput transaction processing (OLTP) system. For example, prior work has shown that 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 logging [11].

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 our work on the H-Store [35] system, and in other high throughput data processing systems like RAMCloud [28] and Redis [34], 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.

Recovery

It is also relevant to look at recovery times for the two logging approaches. One would expect ARIES to perform recovery faster than command logging; because in command logging, transactions need to be re-executed completely at recovery time whereas in ARIES, 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.

1.2 Why the Data Durability Problem Needs to be Revisited

As CPUs have gotten faster compared to disks and main memory capacities have grown, many OLTP applications have become memory-resident and thus the run-time of an OLTP transaction has shrunk. However, logging times have not improved as dramatically, because logging still involves substantial disk I/O and many CPU instructions. As such, in modern OLTP systems, command logging is an increasingly attractive alternative logging approach.

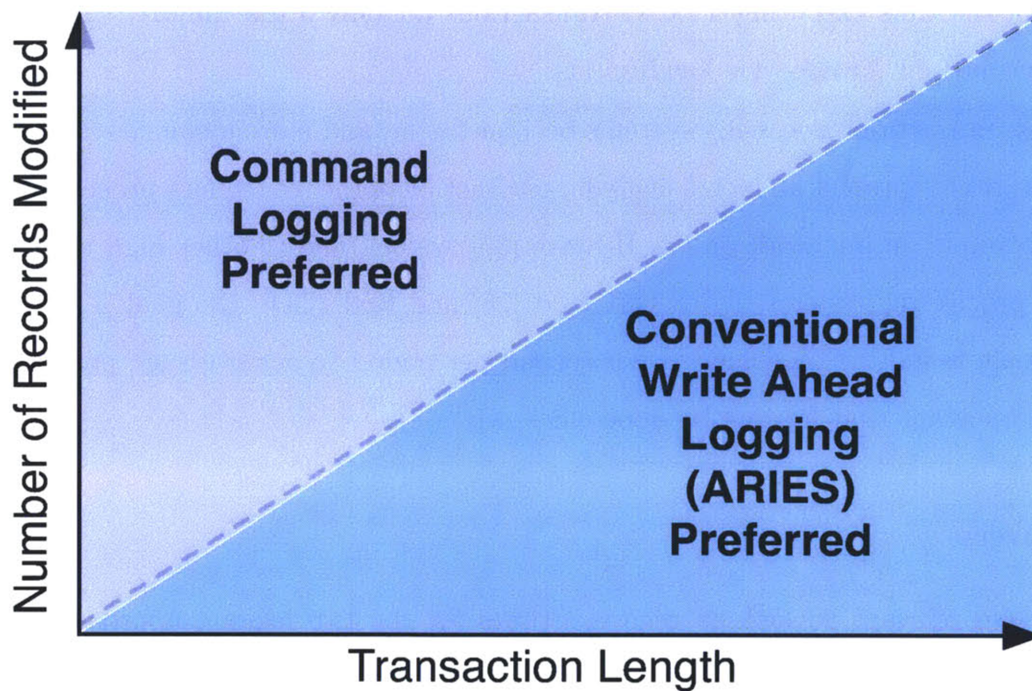


Figure 1-1: Illustration of when command logging is preferred over ARIES.

Figure 1-1 shows a conceptualization of this idea. The region on the right below the diagonal contains both complex transactions involving lots of computation per data write, as well as low-throughput applications. In this region, the overhead of logging in ARIES is small relative to command logging, so it may be preferred due to lower recovery times. In high throughput settings with simple transactions (left region in the plot, above the diagonal), command logging is preferable, especially so when the transactions update many database tuples (leading to lots of logging in ARIES).

Increasingly, the trend in modern transaction processing applications is to be at the left side of Figure 1-1. This is because of two reasons:

1. First, transaction processing applications inherently involve simple, update intensive transactions.

2. Second, as RAM capacities of modern machines continue to grow ever larger, transaction processing is becoming memory resident, meaning that transaction latencies are becoming shorter and shorter.

Even systems not tuned for high throughput, like MySQL and Postgres, are now capable of running real-world transactional applications at throughputs of thousands of transactions per second, while specialized systems like H-Store and RAMCloud report numbers that are an order of magnitude higher.

1.3 Thesis Contributions

In this thesis, our goal is to study the discussed performance trade-offs between ARIES 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 ARIES in the VoltDB main memory open-source database system [37] (VoltDB is based on the design of H-Store [35]).

We compare the performance of both the logging modes on two transactional benchmarks, Voter and TPC-C.

1.3.1 High-level Overview

Our experimental results show that command logging has a much lower run-time overhead compared to ARIES when:

(a) transactions are not very complex, so that disk I/O due to logging represents a substantial fraction of transaction execution time, and

(b) the ratio of command log record size to number of data tuples updated by a transaction is small, because ARIES does much more work in this case.

For example, TPC-C has short transactions that update a moderate number of records and it falls in the left region of Figure 1-1. In our experiments, we found that command logging can achieve about 1.5× the throughput of ARIES for TPC-C when running at maximum throughput (about 4K transactions per second (tps) per core), a result which is in line with the plot's prediction. Also for TPC-C, we found that

recovery times, as expected, are better for ARIES than command logging, by a factor of about 1.5.

1.4 Road Map

The rest of this thesis is organized as follows. We begin with a discussion of VoltDB's system architecture in Chapter 2. We then describe our approach to command logging in Chapter 3, followed by a detailed description of our main-memory adaptation of ARIES in Chapter 4. Subsequently, we report extensive performance experiments in Chapter 5 and discuss possible approaches to generalize command logging in Chapter 6. Chapter 7 provides an overview of relevant past work in this area, and Chapter 8 summarizes the thesis.

Chapter 2

VoltDB Overview

VoltDB is an open source main memory database system whose design is based on that of the H-Store system [35], with some differences. In this thesis, we compare performance of ARIES and command logging in VoltDB.

2.1 Architecture

In VoltDB, the database is horizontally partitioned on keys and these partitions reside in main memory of nodes in a cluster. Partitions are replicated across different nodes for high availability. All associated indexes are also kept in main memory along with the partitions. Every node in the cluster runs multiple execution sites (e.g., one per CPU core), and each partition residing on the node is assigned to only one such site. Each cluster node has an *initiator* site which sends out transactions to the appropriate partition(s)/replicas.

By employing careful, workload-aware partitioning, most transactions can be made *single-sited* (run on just a single partition) [31].

2.2 Transactions as Stored Procedures

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, and each transaction is a single stored procedure. Although this scheme requires all transactions to be known in advance, 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.

2.3 Concurrency Control and Durability

Because OLTP transactions are short, typically touching only a small number of database records and do not experience application or disk stalls, transactions for a given partition can simply be executed in serial order on that partition, often without any concurrency control at all [15].

These mechanisms 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 provide availability of database contents. VoltDB uses command logging (described in Chapter 3), 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.

2.3.1 Checkpoint Mechanism

VoltDB's checkpoint mechanism periodically writes all committed database state to disk, but index updates are not propagated to the disk. At the start of a checkpoint, the database executor goes into a *copy-on-write* mode. Next the snapshotting thread goes through memory, using the dirty bit to determine whether to snapshot a tuple or back it up in the shadow table. When one such sweep is done, the executor returns to regular mode. Thus, there is no need to quiesce the system and the checkpoint can be written asynchronously in the background. The checkpointing is asynchronous in

order to prevent checkpoints from being slow.

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 that transaction isolation is implemented in snapshot-isolation based systems.

Given this overview of VoltDB, we next describe how our implementation of command logging in VoltDB works.

Chapter 3

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 Chapter 1, the advantage of command logging is that it is extremely lightweight, requiring just a single log record to be written per transaction.

3.1 Logging Transactions as Stored Procedures

For the rest of this thesis, 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 command logging then consist of the name of a stored procedure and the parameters to be passed to the procedure.

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

mand logging does not support transaction *savepoints* [26]. As OLTP transactions are short, this is not likely to be a significant limitation.

3.1.1 Writing a Log-record

Writing a log record to the command log for a new single-node transaction is relatively straightforward. However for a distributed transaction, a write to the command log is more involved, requiring a two-phase commit-like protocol to ensure that participating sites have accepted the transaction. In our VoltDB implementation of command logging, this write is done as follows.

At the beginning of a transaction, the coordinator site (specific to this transaction as there is no global cluster coordinator) sends out the transaction to all sites participating in the transaction and to other replicas of itself. Each execution site writes the transaction to its command log on disk; note that multiple execution sites on the same node write to a shared command log. Once a command C is in the log of a site S, S guarantees that it will execute C before any other commands that appear after C in S's log. A distributed agreement protocol based on special queues per execution site is used to agree on a single global order of execution of transactions that all partitions and replicas see. Even if one or more sites do not respond to the coordinator (e.g., because of a crash), the transaction will be executed as long as a covering set of replicas is available. The cluster halts when there is no replica covering a partition.

3.1.2 Optimizing for Performance

Command logging can either be synchronous or asynchronous. When the command logging is asynchronous, the database server can report to the transaction coordinator that the transaction committed even if the corresponding log record has not yet propagated to disk. In the event of a failure however, asynchronous logging could result in the system losing some transactions. Hence, we focus on synchronous logging, where a log record for a transaction is forced to disk before the transaction is acknowledged

Checksum	LSN	Record type	Transaction-id	Site-id	Transaction type	Parameters

Figure 3-1: Command logging record structure.

as committed.

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.

3.1.3 Log-record Structure

In our VoltDB implementation of command logging, log records written out have the structure shown in Figure 3-1.

3.1.4 Transaction Rollbacks

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. This list 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.

3.2 Recovery

Recovery processing for a command logging approach is straightforward. First, the latest database snapshot on disk is read to initialize database contents in memory. Because the disk snapshot does not contain indexes, all indexes must be rebuilt at start-up (either parallelly or after the snapshot restore completes).

3.2.1 Transaction Replay

Once the database has been loaded successfully and the indexes have been rebuilt, the command log is read. Because there are multiple execution sites per VoltDB server node, one of the sites takes the role of an initiator and reads the shared log into memory in chunks. Starting from the log record for the first transaction not reflected in the snapshot the database was restored from, log entries are processed by the initiator site. For a command log, one log record corresponds to a single transaction, and as log entries are processed, each transaction is dispatched by the initiator to the appropriate site(s). This ensures the same global transaction ordering as the pre-crash execution, even if the number of execution sites during replay is different from the number of sites at run-time.

3.2.2 Correctness

This recovery process is guaranteed to be correct in VoltDB because (a) transactions are logged and replayed in serial order, so re-execution replays exactly the same set of transactions as in the initial execution, and (b) replay begins from a transactionally-consistent snapshot that does not contain any uncommitted data, so no rollback is necessary at recovery time.

We discuss how command logging could be extended to other database systems in Chapter 6.

Chapter 4

ARIES for Main Memory

ARIES [26] was originally intended as a recovery algorithm for a disk-based database system. In a traditional ARIES style data log, each operation (insert/delete/update) by a transaction is written to a *log record table* before the update is actually performed on the data. Each log entry contains the before and after images of modified data. Recovery using ARIES happens in several passes, which include a physical REDO pass and a logical UNDO pass.

While the core idea behind traditional ARIES can be used in a main-memory database, substantial changes to the algorithm are required for it to work in a main memory context. In addition, the main-memory environment can be exploited to make ARIES logging more efficient.

We discuss both these aspects to main-memory ARIES in detail below.

4.1 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 [33]). These log records contain disk specific fields such as 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.

4.1.1 Identifying a Modified Tuple

In an 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, ARIES features optimized for disk access such as the various buffer pool and disk mechanisms can be stripped off. Specifically, all disk related fields in the log record structure can be removed. For each modification to a database tuple, we can simply write 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 5.1 for schema details).

Note that the virtual address of a modified in-memory data tuple is not a good choice as a unique identifier for two reasons: (1) the database periodically performs compaction over the table memory layout to minimize data fragmentation, (2) the database is not guaranteed to load a table and all its records at the same virtual address on restart after a crash.

Much like disk-based ARIES uses RIDs to identify the location of a change before it re-applies an update at recovery time, in-memory ARIES uses `(table-name, primary-key)` to identify the data tuple's location at recovery time, and the serialized after-image bytes in the log are used to modify the in-memory data tuple appropri-

ately. For the primary-key lookup identifying a tuple's location to be fast, an index on the primary key must be present at replay time. In VoltDB, index modifications are not logged to disk at run-time, so all indexes must be reconstructed at recovery time prior to replay.

Wide Tuples

For tables with wide rows, a large amount of ARIES log space can be saved by additionally recording which columns in the tuple were updated by a transaction, with before and after images for only those columns instead of the entire tuple (this optimization does not apply to inserts). We found that that this optimization led to a significant reduction in a log record's size for the TPC-C benchmark.

4.1.2 Dirty Record Table

An in-memory database has no concept of disk pages, and thus ARIES does 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, such a table can grow fairly large within a short duration of execution; regular snapshotting would keep the table size bounded however. 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.

Disk-based ARIES assumes fuzzy checkpointing [21] to be the database snapshot mechanism. Fuzzy checkpoints happen concurrently with regular transaction activity, and thus updates made by uncommitted transactions can also get written to disk as a part of the checkpoint. Both the dirty page and transaction tables are flushed to disk along with each checkpoint in disk based ARIES, 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 each ARIES checkpoint would need to scan the in-memory database to construct the *dirty record table* before it can be written along with the checkpoint.

Alternatively, we could use transaction-consistent checkpointing [32] instead of fuzzy checkpointing. VoltDB already uses non-blocking transaction-consistent checkpointing (see Chapter 2), so we leveraged it for our ARIES implementation. With transaction consistent checkpointing, only updates from committed transactions are made persistent, so that ARIES can simply keep track of the oldest LSN whose updates have not yet been reflected on disk. Thus, the dirty records table can be eliminated altogether for this checkpointing scheme.

4.1.3 Other Optimizations

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. Our implementation of ARIES does not support partial rollbacks or *savepoints*, so that this optimization can be applied without repercussions.

ARIES has traditionally been a synchronous logging technique in the sense that log writes of a committed transaction are forced to disk before the server reports back the transaction's status as committed. Similar to command logging, ARIES uses group commit; writes from different transactions are batched together to achieve better disk throughput and to reduce the logging overhead per transaction.

4.1.4 Log-record Structure

In our VoltDB implementation of ARIES, execution sites on the same node write to a shared ARIES log with arbitrary interleaving of log records from different sites. The ordering of log records per site is still preserved. A field in the ARIES log record identifies the site/partition to which the update corresponds. The log record structure

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

Figure 4-1: ARIES log record structure.

for our ARIES implementation in VoltDB is shown in Figure 4-1.

4.2 Recovery for Main-memory

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 allows for independent recovery of different database objects and parallelism during recovery. The undo pass in ARIES is *logical*.

4.2.1 REDO Phase

ARIES recovery in a main-memory database also begins with the 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 and the serialized after-image bytes in the log record are used to modify the tuple (this covers insert and delete operations as well). 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 Chapter 5 for performance numbers).

4.2.2 UNDO Phase

After the redo pass 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).

ARIES 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 site, only the transaction executing at the time of crash will need to be rolled back (at most one per site).

Optimizing UNDO

Even a single rollback per site 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. This makes it possible to reduce log record sizes by nearly a factor of two, as the before image in update records can 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.

Chapter 5

Performance Evaluation

We implemented synchronous logging for both command logging and main-memory optimized ARIES inside VoltDB. Because the logging is synchronous, both logging techniques issue an `fsync` to ensure that a transaction's log records are written to disk before returning the transaction's results. We implemented group-commit for both the logging techniques, with the additional ARIES optimization that 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 ARIES implementation group-commits with the same frequency as command logging.

In this chapter, we show experimental results comparing command logging against ARIES. 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 variation of throughput and latency with group-commit frequency. Results on how throughput and recovery times vary as transaction lengths go up for a fixed amount of logging are also presented.

In Section 5.1, we briefly discuss the benchmarks we used in our study. Then we describe our experimental setup in Section 5.2 followed by performance results in

```
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)
```

Figure 5-1: Voter benchmark schema.

Section 5.3. Finally, we summarize our results and discuss their high level implications in Section 5.4.

5.1 Benchmarks

We use two different OLTP benchmarks in our study, Voter 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 transactions. TPC-C database tables are also much wider and exhibit complex relationships with each other compared to that of the tables in the Voter database.

The two benchmarks are described below.

5.1.1 Voter

The Voter benchmark simulates a phone based election process. The database schema is extremely simple and is shown in Figure 5-1.

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 extremely simple 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 values of 6 and 2 respectively.

5.1.2 TPC-C

TPC-C [36] 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.

Our implementation of TPC-C does not take think times into account.

5.2 Experimental Setup

In all our experiments, the database server was run on a single Intel Xeon 8-core server box with a processor speed of 2.4GHz, 24GB of RAM, 12TB of hard disk in a RAID-5 configuration and running Ubuntu Server 10.10. To improve disk throughput without sacrificing durability, the disk was mounted as an ext4 filesystem with the additional flags {`noatime,nodiratime,data=writeback`}. Note that ext4 has write barriers enabled by default which ensures that `fsyncs` are not acknowledged until the data hits the disk. However, our disk had a battery backed controller so that `barrier=1` was ignored by the filesystem. For both

ARIES and command logging, log files were pre-allocated to prevent additional seeks resulting from the OS updating the file metadata while flushing the log to disk.Such a careful setup is necessary to optimize either recovery system.

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 5 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.

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

5.3 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 ARIES turned off, (b) ARIES logging on and command logging turned off, and (c) both command logging and ARIES turned off.

5.3.1 Throughput

Figure 5-2 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, ARIES-logging and command-logging) are able to match the client rate until 80K tps at which ARIES 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 during the transaction, ARIES suffers about 15% drop in maximum throughput at run time. For more complex transactions, the ARIES performance penalty is higher as we see next.

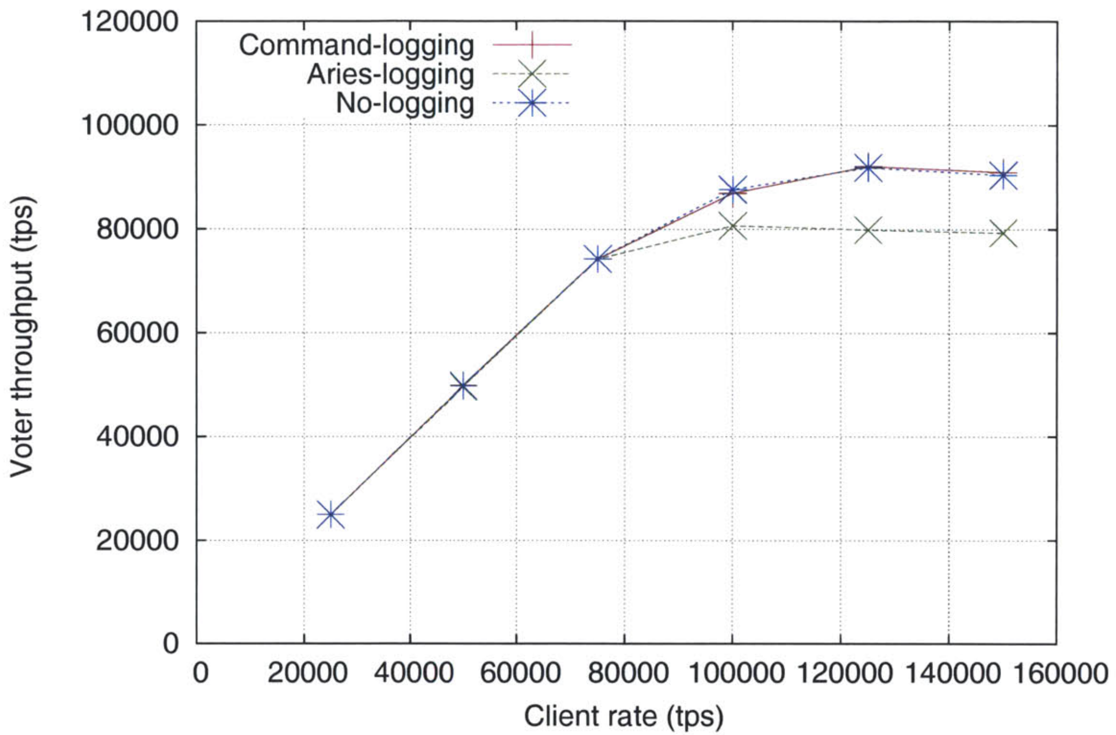


Figure 5-2: Voter throughput vs. client rate (both tps).

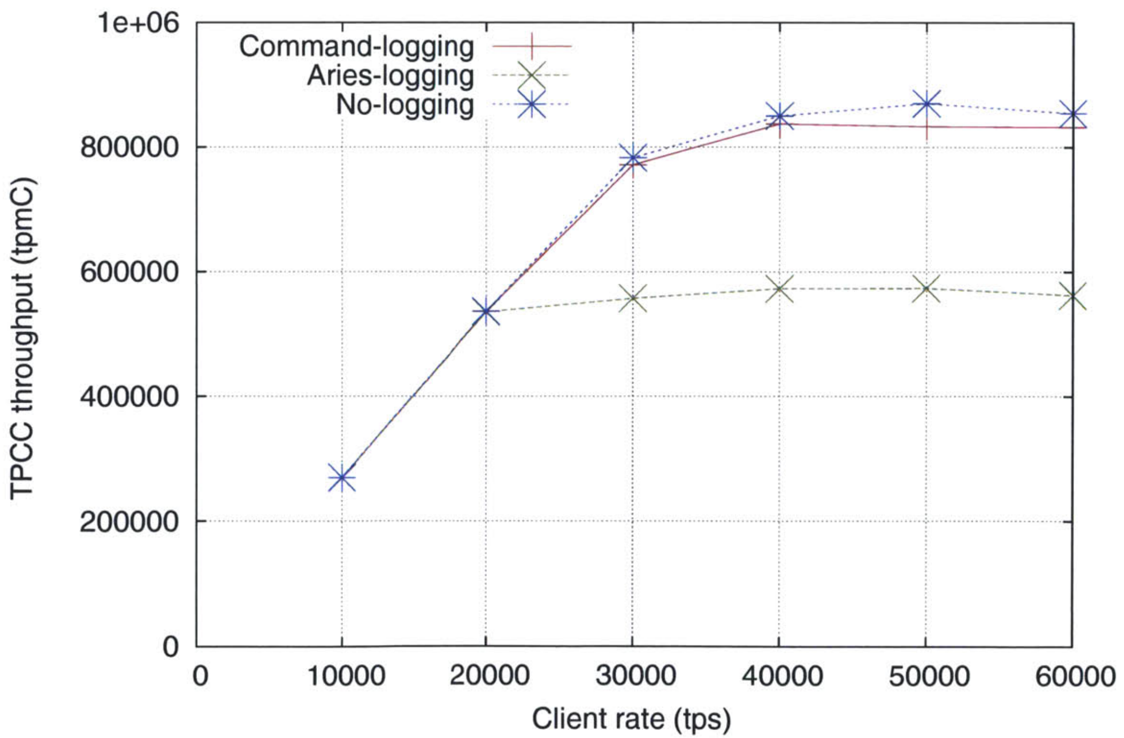


Figure 5-3: TPC-C throughput (tpmC) vs. client rate (tps).

Figure 5-3 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 ARIES 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 ARIES 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 5.3.3 for numbers). The penalty on Voter is lower because the number of log writes for the `vote` transaction is small (just one).

Using the two dimensions of Figure 1-1, both approaches have short transactions, and operate in the "command logging preferred" region of the graph, but TPC-C performs more updates per transaction, and is favored more heavily by command logging.

5.3.2 Latency

The variation of transaction latency with client rates for the voter benchmark is shown in Figure 5-4. 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 (the effect of varying group commit frequencies is studied in a later experiment). The latencies for all methods gradually increase as the database server approaches saturation load. Command-logging has almost the same latency as no logging and ARIES's latency is about 15% higher. 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.

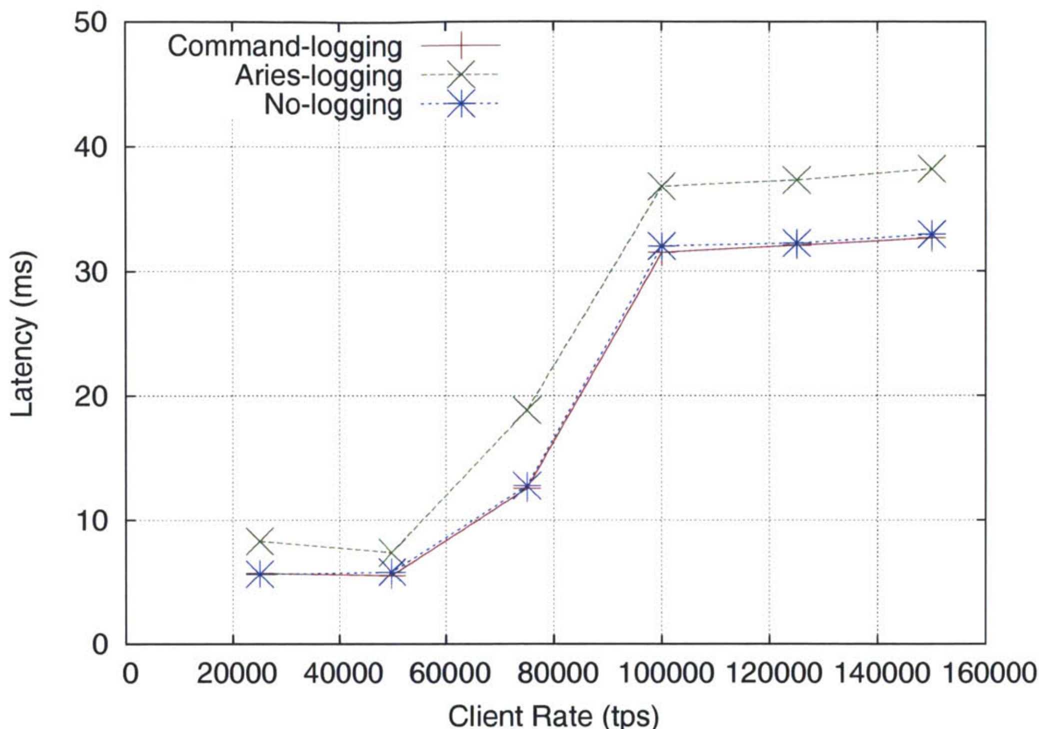


Figure 5-4: Voter latency in milliseconds vs. client rate (tps).

In Figure 5-5, we see that TPC-C performs similarly, except that ARIES reaches saturation at about 21K tps, so that its latency also jumps higher 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, ARIES latencies are consistently at least 45% higher for all client rates.

5.3.3 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, ARIES 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

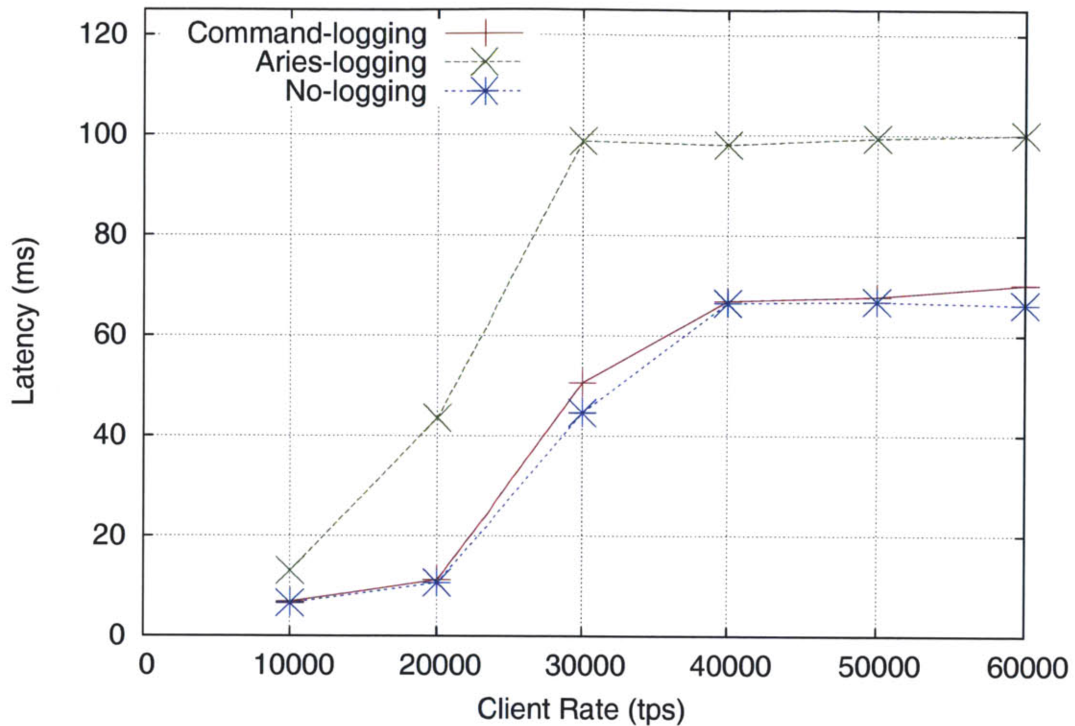


Figure 5-5: TPC-C latency in milliseconds vs. client rate (tps).

not exist. Moreover, as discussed in Chapter 4, 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. Figure 5-6 shows on a log scale the number of rows affected in each table for the different operations. 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.

Whenever an insert, delete or update is performed on a database tuple, ARIES writes a log record identifying the tuple, the table in question as well as the operation (along with a log header). In case of an insert, the entire tuple contents are recorded

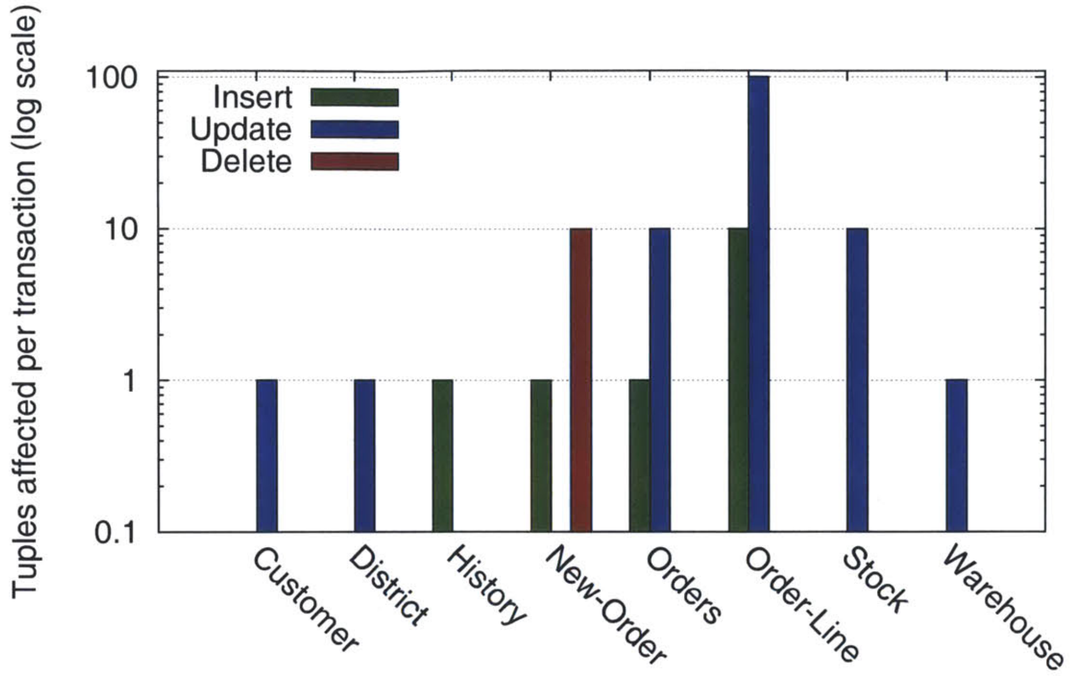


Figure 5-6: Number of data tuples modified per transaction for different TPC-C tables. Note that the y-axis is on a log scale.

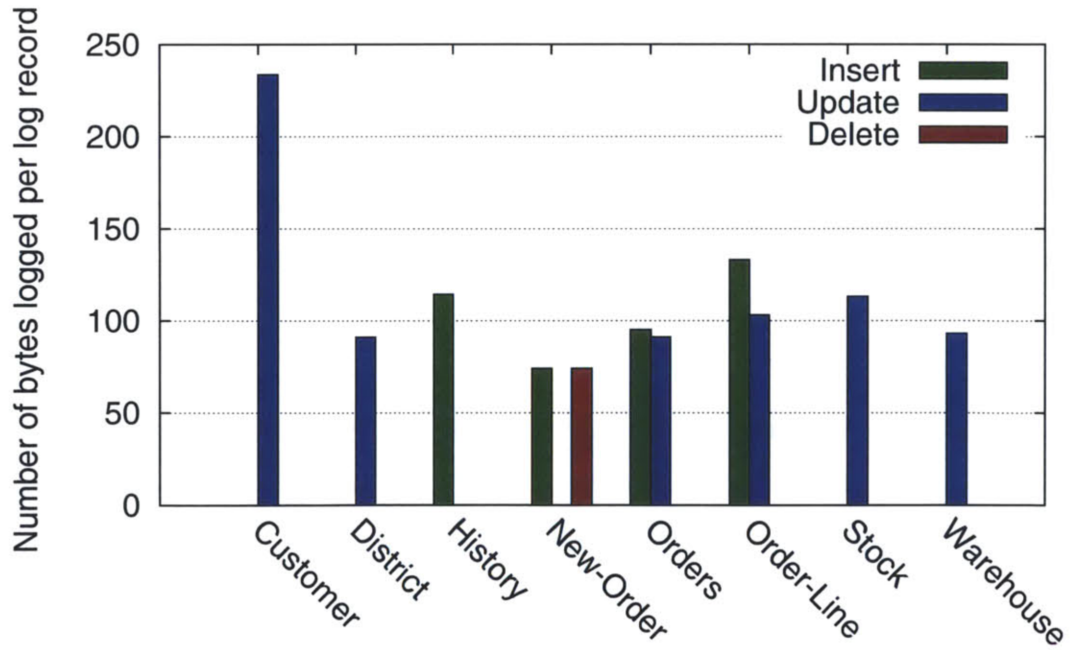


Figure 5-7: Number of bytes written per ARIES log record for different TPC-C tables.

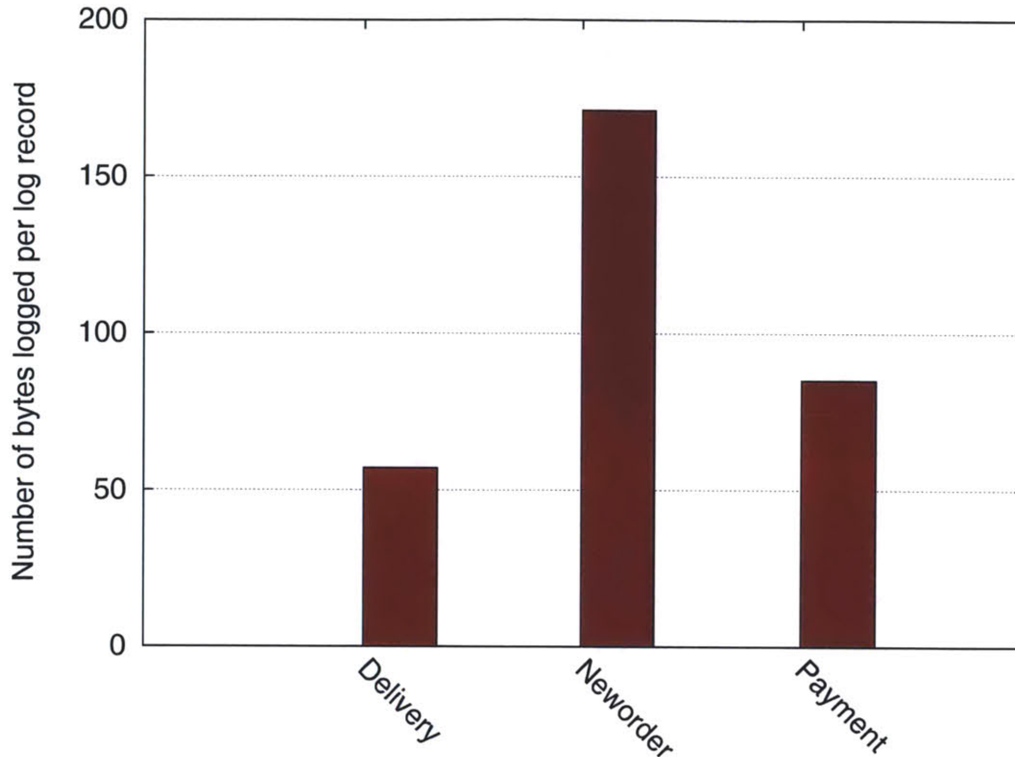


Figure 5-8: Number of bytes written per log record by command logging for different stored procedures.

in the after image. For an update, only modifications to the tuple are recorded. Depending on the table that is updated, ARIES log record sizes vary from 70 bytes (New-Order table) to 240 bytes (Customer table) per record, with most log records being less than 115 bytes as can be seen in Figure 5-7.

For command logging, Figure 5-8 shows that the three transactions write between 50 (*delivery*) and 170 (*neworder*) bytes per transaction (there is only one log record for each 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.

Overall, on TPC-C, in comparison to command logging, ARIES logs about 10× more data per transaction (averaged over the three transaction types).

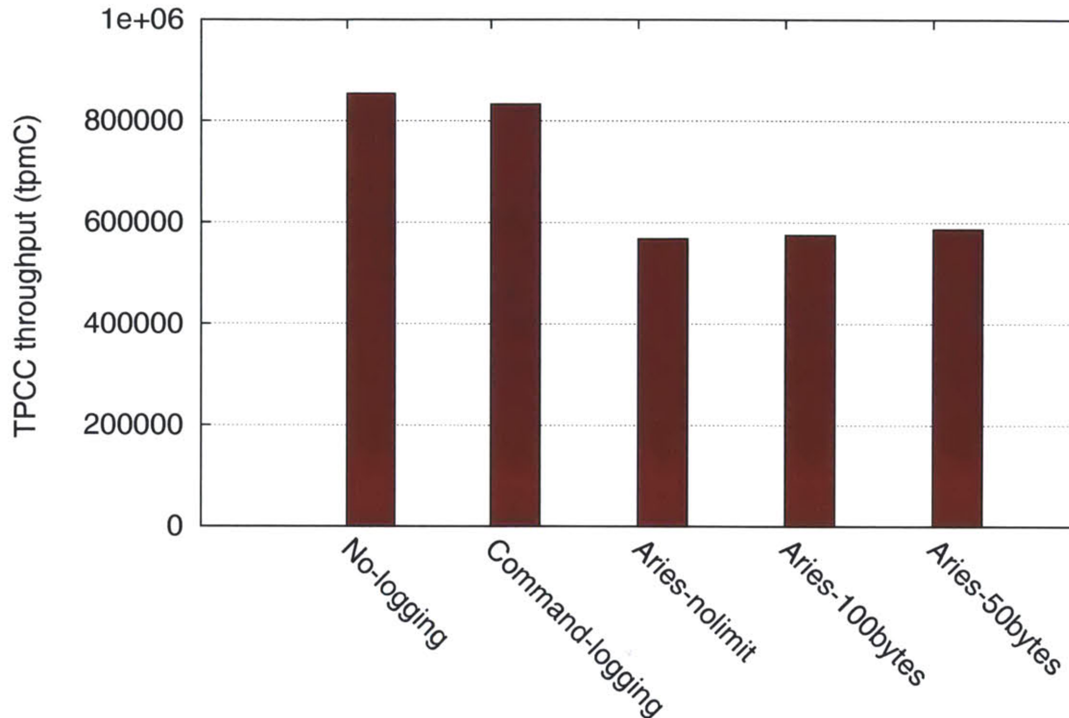


Figure 5-9: Logging mode's TPC-C throughput does not merely depend on the number of log bytes written to disk.

5.3.4 Log Record Size vs. Performance

Because ARIES writes so much more data than command logging on TPC-C, we wanted to test to see if the run-time performance difference between the two systems on this benchmark was completely attributable to I/O time.

We ran an experiment to see if this was the case by truncating the size of ARIES log records written out per transaction to a fixed number of bytes. The resulting recovery log is unrecoverable/corrupt, but this is not important for the purposes of this experiment.

Figure 5-9 shows the results. In addition to bars for the three logging modes, we have bars for two synthetic cases where ARIES does the same amount of CPU work creating log records but only writes a maximum of either 50 or 100 bytes to the log for the entire transaction (here 100 bytes is approximately what command logging writes on an average for a TPC-C transaction). In both cases, we see that ARIES throughput slightly increases yet remains lower than command logging by nearly the

same factor.

Thus, the performance gap at run-time between command logging and ARIES is a result of not only the extra disk I/O that ARIES needs to do to write larger records to disk, but also of the higher CPU overhead incurred in logging activities during transaction execution.

5.3.5 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, then rebuilding all indexes and finally 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 ARIES 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 5-10 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. 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 5-2). On the other hand, ARIES is able to replay the log almost 5× faster at about 500K tps. This difference is due to the fact that ARIES 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 ARIES overhead of parsing each log record and reapplying the relevant updates is small.

In Figure 5-11, we see that even for the TPC-C benchmark, ARIES log replay is faster compared to command logging. Command logging can only recover at about

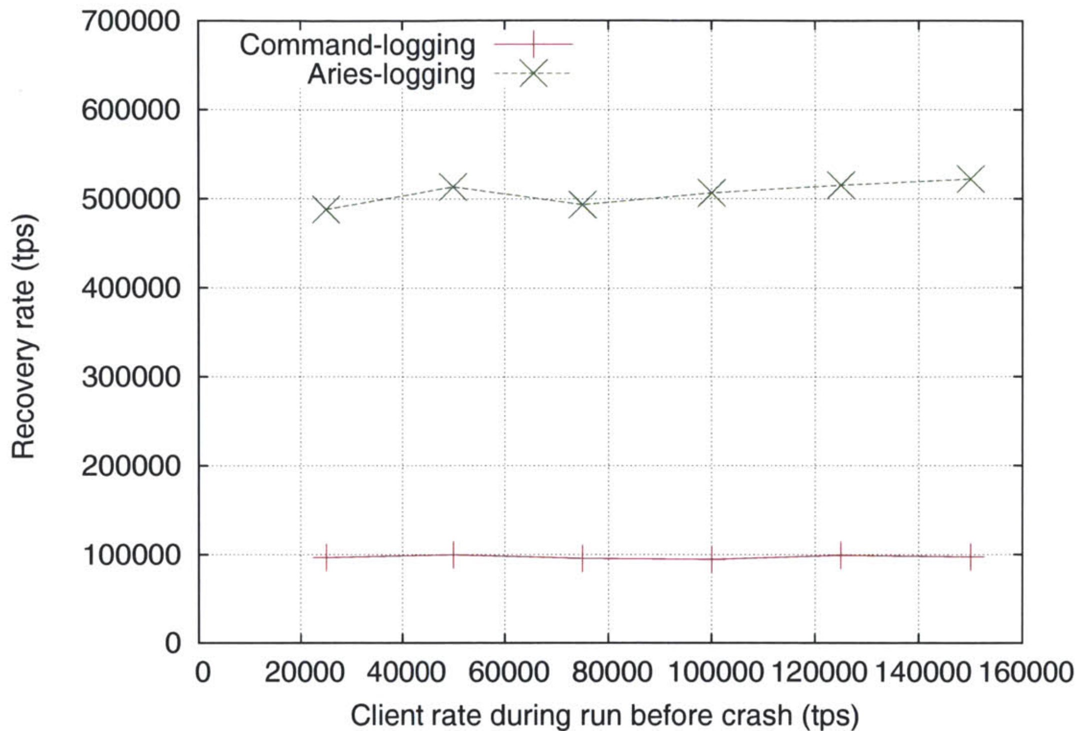


Figure 5-10: Voter log replay rates (tps).

865K tpmC, which is also its maximum run-time throughput on an average (Figure 5-3). However owing to the increased complexity of TPC-C transactions, ARIES must now pay a much higher cost of parsing its more sophisticated log records for each transaction compared to command logging. This is reflected in the reduced gap between command logging and ARIES recovery speeds; ARIES replay is only about 1.5× faster for TPC-C as opposed to the 5× speedup for voter.

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 ARIES.

For both benchmarks, the log read in case of command logging added less than 1% extra overhead to the replay time. The reasons for this were two-fold: (a) the amount of data written per transaction by command logging is rather small (as seen earlier in Section 5.3.3), and (b) re-execution of transactions is expensive compared to the

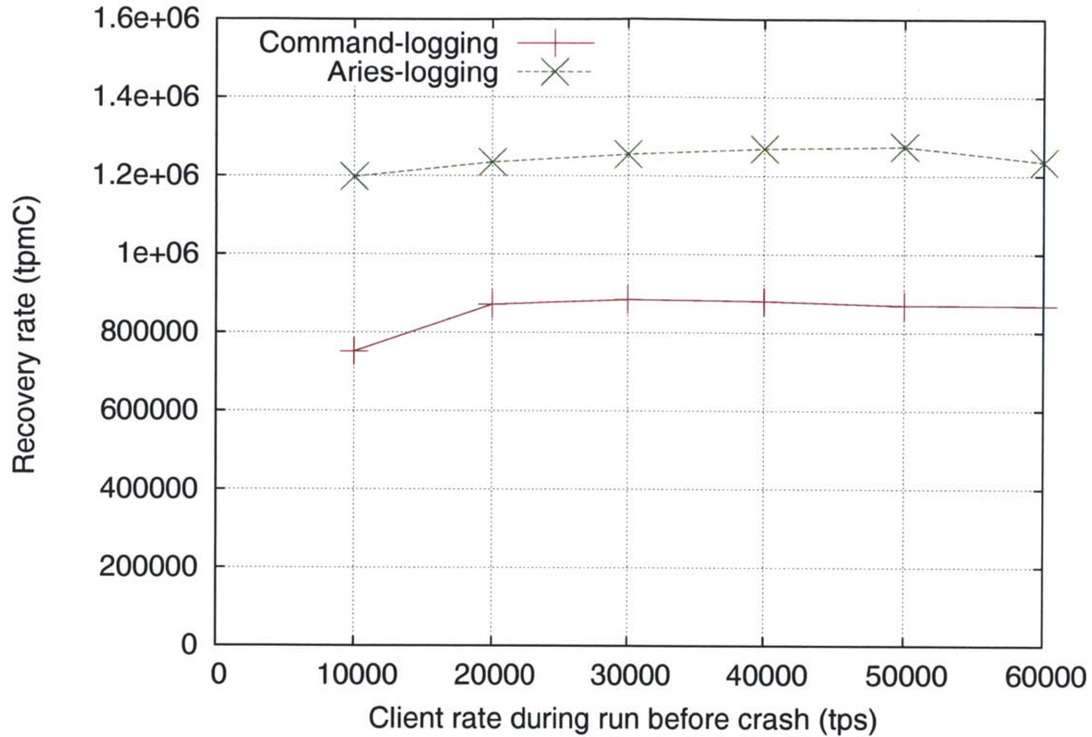


Figure 5-11: TPC-C log replay rates (tpmC).

amortized I/O cost of reading a transaction's log records. In contrast, the numbers look very different for ARIES; log read now accounted for an additional 30% overhead to the replay time for voter and about 8% added overhead to the replay time for TPC-C. Because ARIES replay can happen in parallel, it appears that a single core read of the log adds to the faster ARIES replay times substantially; the high 30% added overhead for the simpler voter benchmark attests to this.

5.3.6 Group Commit Frequency

The idea behind group commit is to batch together log records for different transactions and flush them to the disk together. In this way, synchronous disk writes are amortized across many transactions, at the cost of additional latency per transaction. Group commit is essential for obtaining high throughput from a transactional database system that employs logging. It is important that both command logging and ARIES group-commit at the same frequency for an apples-to-apples comparison.

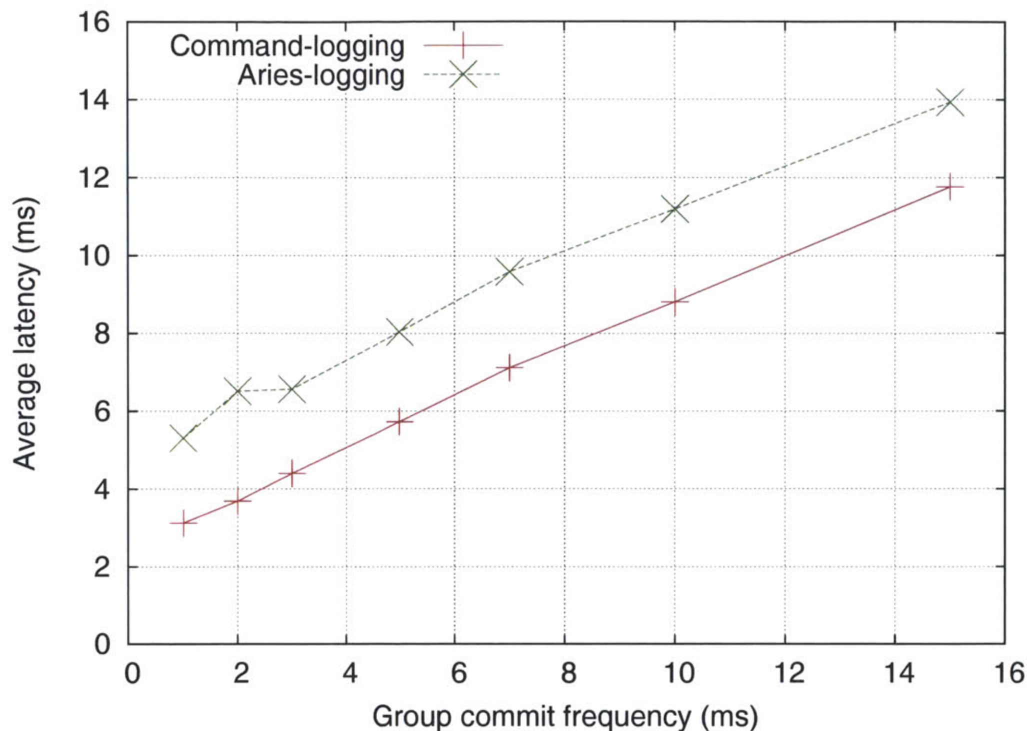


Figure 5-12: Voter latency (ms) vs group commit frequency (ms).

We ran an experiment to study the effect varying group commit frequency has on transaction latency when the server is running well below maximum capacity. This is important so that the increased wait latency for a saturated server does not mask the effect that group commit frequency has on latency. From Figures 5-12 and 5-13, we see that for both voter and TPC-C, as the group commit frequency is varied from 1 ms to 15 ms, average latencies for ARIES and command logging increase linearly. It appears that larger group commit frequency values linearly hurt latency.

To observe the effect of group commit frequency on system throughput, we ensured that the database servers were saturated otherwise throughput will not depend on group commit frequency.

For voter, because each transaction is extremely short and touches only one database record, every millisecond of extra time spent waiting for group commit decreased throughput as the frequency varied from 1 ms to 15 ms. At a group-commit frequency of 15 ms, throughput was down by 10% compared to the saturation

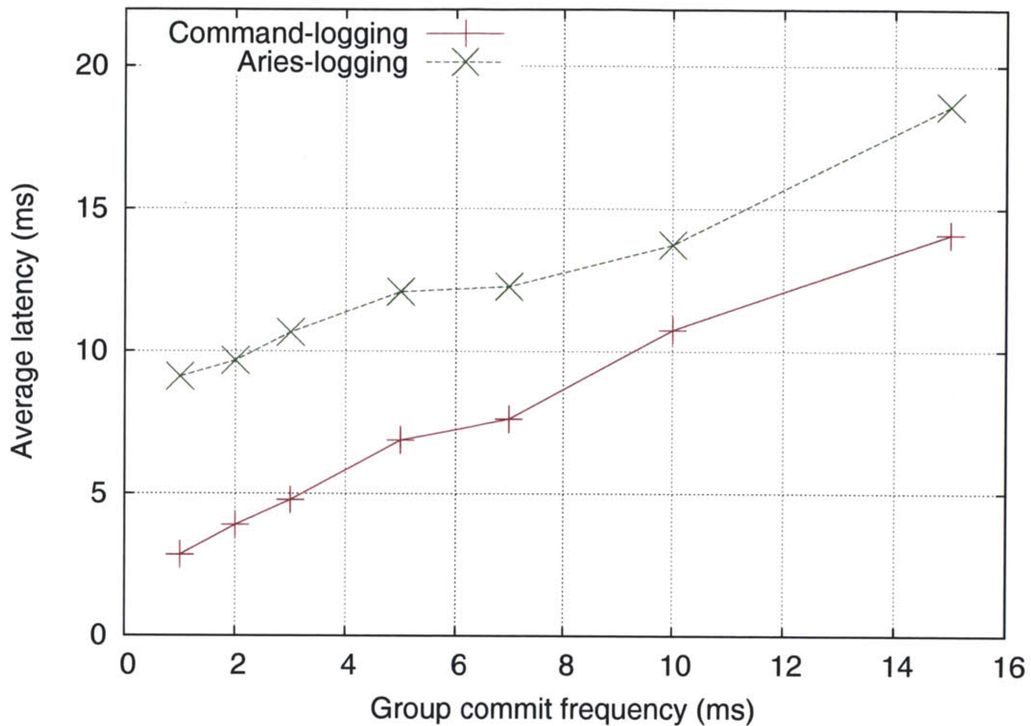


Figure 5-13: TPC-C latency (ms) vs group commit frequency (ms).

throughput we saw in Figure 5-2.

TPC-C transactions are more complex and we observed that varying the group commit frequency from 1 ms to 15 ms had negligible effect on both command logging and ARIES TPC-C throughputs. In fact, for extremely high frequency flush rates of 1 ms to 2 ms, we saw that the performance actually took a slight dip, suggesting that we may be hitting hardware limits.

Based on the effect of group commit frequency on latency and throughput for both benchmarks, we believe that setting the group commit frequency to a 3 ms-5 ms value leads to good performance.

5.3.7 Transaction Length

As we noted in Chapter 1, OLTP transactions have become shorter as processors have gotten faster and RAM sizes of tens of gigabytes have become routine. To simulate slower machines and longer OLTP transactions of the era in which ARIES

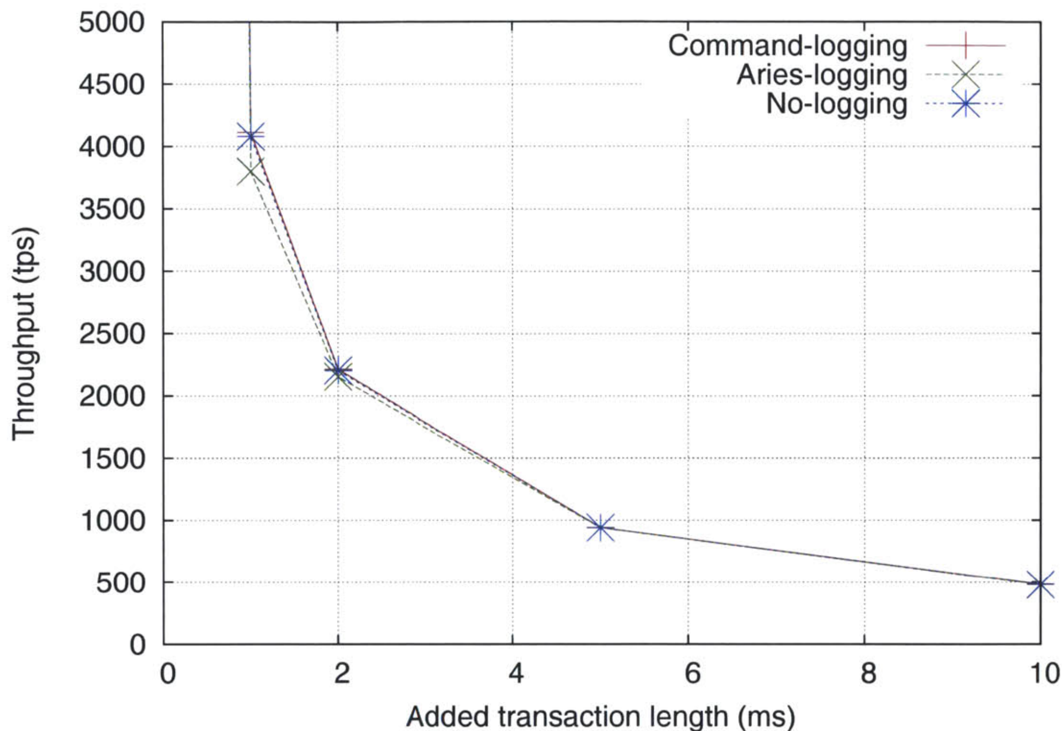


Figure 5-14: Voter run-time throughput (tps) vs. added transaction length (ms).

was conceived, we added extra computation to transactions in both our benchmarks while keeping the amount of logging work done by the transaction the same as before. The extra computation increased the transaction length by a fixed duration, in the range 1 ms-10 ms. Our hypothesis is that a longer transaction length should make ARIES look better, because logging will represent a small fraction of the total work the transaction does.

Figures 5-14 and 5-15 show how the database throughput varies for no logging and the two logging modes as the transaction lengths go up for voter and TPC-C. For all data points on these two plots, the database server was running at saturation throughput. The 0 ms point at the extreme left on the plots corresponds to normal voter/TPC-C transactions where no extra computation was added, but is not shown because including it makes the graph illegible (throughput with no added computation is shown in Figures 5-2 and 5-3).

We see that even with just 1 ms of added transaction length, the throughput for

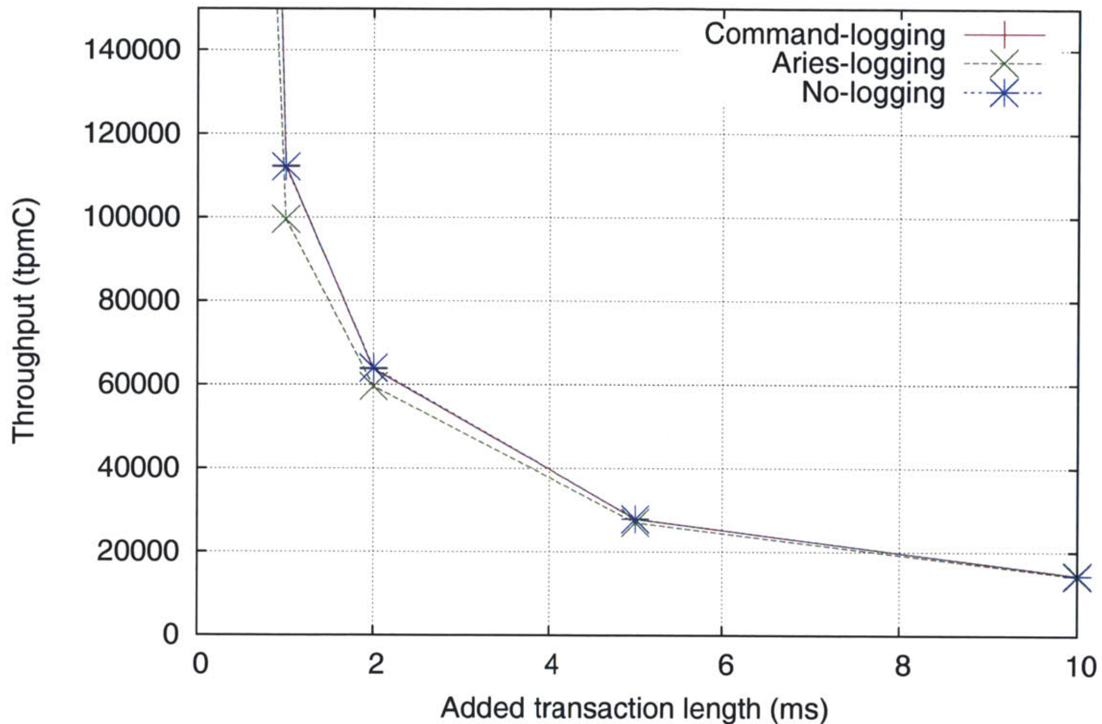


Figure 5-15: TPC-C run-time throughput (tpmC) vs. added transaction length (ms).

the voter benchmark drops by a factor of 20 for ARIES and about a factor of 24 for the other two logging modes. At 1 ms added transaction length, ARIES now takes less than a 5% hit in performance compared to command logging. This performance gap reduces to about 1% as the transactions become longer than 10 ms. The added transaction length in effect amortizes the ARIES CPU overhead, resulting in nearly same throughputs for all logging modes.

TPC-C transactions are more complex and run longer than voter transactions, so that the initial performance drop when 1 ms of extra computation is added is less drastic compared to voter for all logging modes. With a factor of 8 drop for no-logging and command logging and about a factor of 5 drop for ARIES, ARIES throughput already gets within 12% of the throughput achieved by command logging with just 1 ms of added transaction length. This gap in performance narrows down to less than 2% as the added transaction lengths reach 10 ms. The explanation for this behavior is the same as for the voter benchmark: relative to the transaction's

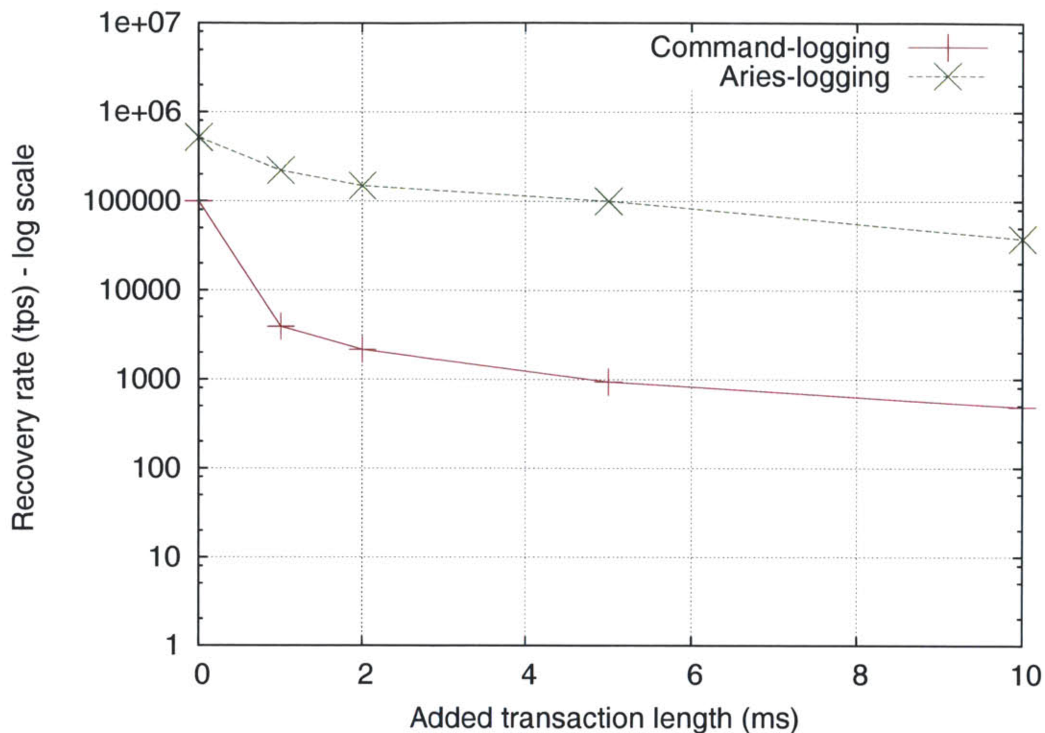


Figure 5-16: Voter replay rates (tps) on log scale vs. added transaction length.

work, ARIES logging represents a much smaller fraction of the overall work.

Even though transaction lengths are longer at run time, recovery times are not necessarily affected by this increased computation. In particular, ARIES recovery can simply read off all updates without redoing the extra computation and apply them to the database correctly. However, a replay of the command log takes much longer because the long transactions must actually be re-executed. Thus, we expect ARIES to be a clear winner at recovery time for longer transactions. Figures 5-16 and 5-17 attest to this being the case for both voter and TPC-C. Note that both the plots are on a log scale. As expected, because the transactions now involve extra computation, command logging recovery rates drop by orders of magnitude.

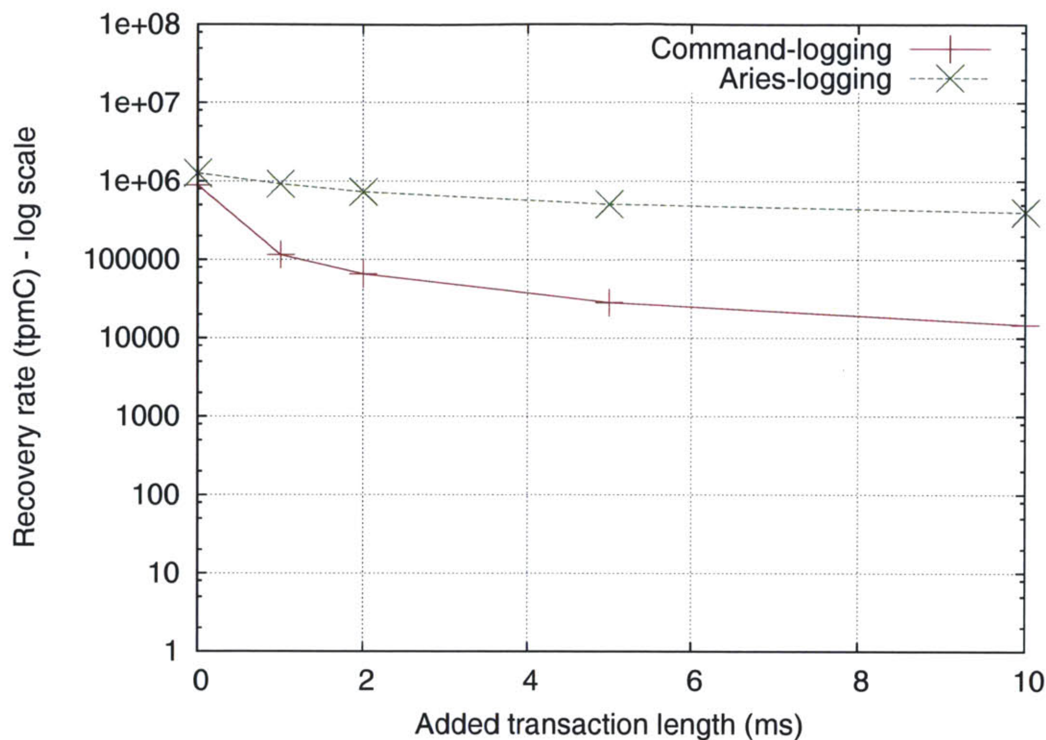


Figure 5-17: TPC-C replay rates (tpmC) on log scale vs. added transaction length (ms).

5.4 Discussion

Our results shows that command logging has a much lower run-time overhead than ARIES (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 ARIES 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 ARIES 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

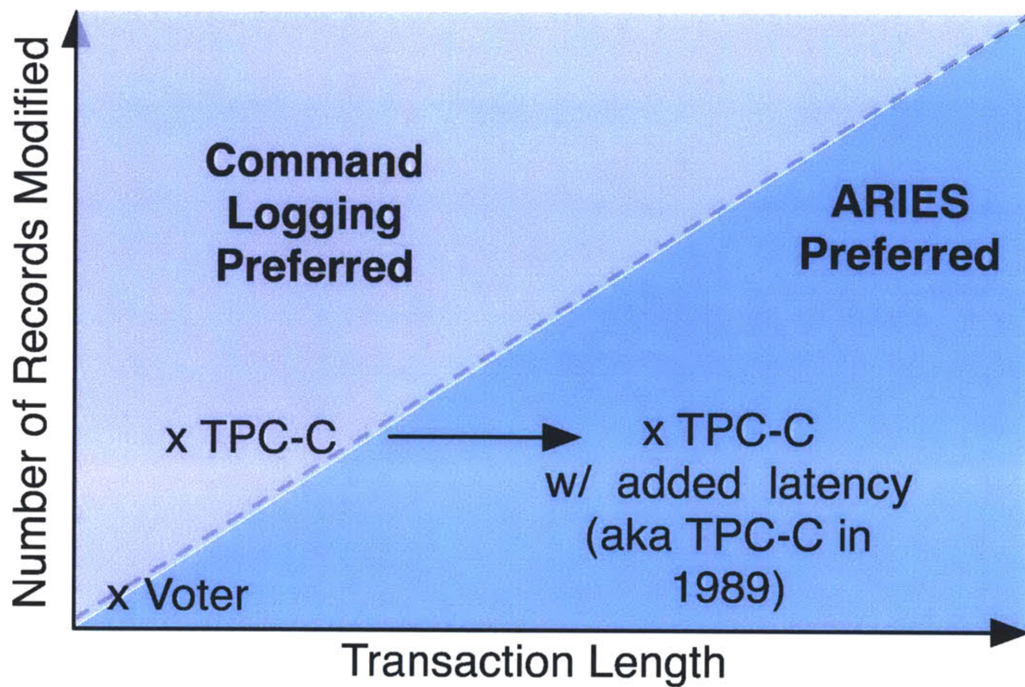


Figure 5-18: Illustration of when command logging is preferred over write-ahead logging, with experimental results overlaid.

overhead at run-time and modest reduction in recovery times, is the best choice.

In our experiments with increased latency per transaction, ARIES does better, since the overheads represent a small fraction of overall run-time, and recovery times for ARIES become *much* better than for command logging. Hence, for applications with complex transactions that update few records (which is not true of most OLTP applications), ARIES 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 in an annotated version of Figure 1-1 shown in Figure 5-18.

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 recovery time is unusually important for some reason.

Chapter 6

Generalizing Command Logging

A natural question about the command-logging approach described in this thesis is how it would generalize to a traditional disk-based system. We believe it should generalize well. To make it work, we need to ensure two properties:

1. First, command log-based recovery needs to start from a transactionally-consistent snapshot.
2. 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.

6.1 Transactionally-consistent Snapshotting

To ensure the first property, if transactions are short-lived, there should be no need to write dirty (uncommitted) pages 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 pages are 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 [32], is an interesting area for future work.

6.2 Equivalent Transaction Replay Order

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 the use of strict two-phase locking for isolation.

Other transactional isolation protocols, like serializable snapshot isolation (SSI) [2], unfortunately do not guarantee that commit order is the same as the serial equivalent execution order.

Furthermore, it is 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.

Chapter 7

Related Work

ARIES [26] is considered the gold standard method for recovery in traditional disk resident databases. The core idea behind ARIES is to write physical log records to disk, with each log entry recording the old and new images of the page being modified. It is ensured that the log records are written to disk before the data modification is propagated to the database (called *write ahead logging* (WAL)). Recovery after crash proceeds via a physical redo followed by a logical undo of transactions that must be rolled back.

Several variants of ARIES such as ARIES/KVL [23] (key value locking), ARIES/IM [27] (individual index entry locking) and ARIES/LHS [24] (recovery of linear hashing based structures) have been proposed in the past. All these variants support index logging in addition to what ARIES does, however most of them have not been implemented [25].

Recovery techniques proposed for main-memory databases (MMDBs) in the past are similar in spirit to ARIES, though many of them date back prior to publication of ARIES work. We briefly go over these techniques here, a detailed discussion of different methods for logging, checkpointing and reloading in main-memory databases can be found in surveys by Garcia-Molina [8] and Dunham [5].

Dali [12] is a storage manager for main-memory resident databases, where the idea is to map persistent data into the the virtual address space of the database process. Their recovery method [13] provides optimizations for ARIES in main memory. The redo records for a transaction are grouped together in memory and written in serial-

ization order to the global log. The idea is that maintaining private redo logs would reduce contention on the global log tail on disk. Undo logs are written to a volatile undo log ahead of any modification to memory, select undo records are written as a part of database checkpoints and the rest of the records in the undo log in main memory are simply discarded on commit. The system incurs a lot of extra overhead while maintaining several redo and undo logs at the same time (one per active transaction).

Work on main memory database recovery techniques by Eich [6] suggests that transaction updates in MMDBs should use main memory shadow pages instead of applying updates in place. The rationale is that rollbacks of uncommitted transactions would then be possible by simply discarding these duplicate pages. On transaction commit, only after-images of the records modified by the transaction are flushed from the log buffer to the log on disk. With the log size cut by nearly half, recovery times after crash are faster due to a shorter persistent log to replay.

Dewitt et al [4] also suggest compressing the size of the log on disk by writing only new values of modified records to disk. However, this requires presence of stable (non-volatile) memory large enough to hold the in-memory write-ahead log for all active transactions. In the absence of such stable memory storage, they resort to flushing log records of transactions in batches (popularly known as *group commit*). Both logging modes in our system (command logging and ARIES) implement the group commit optimization.

Li et al [20] propose a *logging after writing* (LAW) protocol for writing after-images to the log, in contrast with the well-known WAL protocol for before-images. They also suggest run-time optimizations for reducing log size by using shadow pages for updates. However, they require all shadow updates as well as the log buffer to reside in non-volatile memory. Lehman and Carey's recovery algorithm [17] also requires presence of non-volatile RAM to be able to store the redo/undo log tails. This is an interesting assumption but impractical in the sense that this still isn't available on commodity machines. We do not make such an assumption in our system, the entire main memory contents are considered lost after a crash.

Levy and Silberschatz [19] 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, pages are still considered to be a unit of storage in this work as the authors do not believe that it is possible for a database to be entirely kept in memory. Though VoltDB does not have a concept of pages, we believe that if applicable, this idea would be complementary to ARIES logging in our system. Achieving the same is harder with command logging owing to uncertainty about what pages a stored procedure would touch.

On a database restart, replay of a log can only begin after the database has been reloaded from a recent snapshot on disk. Several database reload algorithms are described in [10], such as ordered reload as well as other algorithms including smart and frequency load which allow the database to go online before the database reload has completed. The latter class of reload algorithms lead to higher system throughout but have similar or sometimes worse reload times [10].

Checkpointing during normal operation of a database may be fuzzy or transaction-consistent (or action-consistent). Unlike transaction consistent checkpointing, uncommitted data be written to disk when fuzzy checkpointing is done. ARIES assumes fuzzy checkpointing, indeed it has been suggested that transaction consistent checkpointing is expensive due to locking issues [8]. In contrast to this observation, the overhead of transaction-consistent checkpointing is minimal in VoltDB because it is non-blocking. One advantage of transaction consistent checkpoints is that they make logging easier, allowing logical logging [8].

A study on concurrency control in memory resident databases [18] points out that the cost of acquiring a lock on a data tuple in a main-memory database is typically of the same order as the cost of retrieving the tuple. Instead, a scheme that dynamically varies the granularity of locking on a per relation basis depending on how much transactions conflict is proposed in their work [18]. Work on the memory-resident storage component of IBM's Starburst project [16] also recommends the use of table level latches to reduce run time overhead of the lock manager [9].

A survey paper by Hector et al [8] similarly suggests that due to the low contention

involved in accessing memory resident data, large locking granules ought to be used by transactions. In the extreme, the lock granule could be the entire database, which is equivalent to running transactions sequentially. A 1984 *massive memory machine* proposal [7] seems to be the first to suggest that in the absence of disk I/O bottlenecks, it may be possible to run short transactions sequentially, without any concurrency control. Doing so completely eliminates concurrency control overheads such as acquiring and releasing locks, as well as the overhead of handling deadlocks. As described in Chapter 2, our system (VoltDB) implements this idea, with transactions running serially on each execution site.

QuickStore[38] is a memory-mapped storage system implemented as a C++ class library designed to give programs efficient access to in-memory persistent objects. In this system, log records for recovery are generated by using a page-diffing scheme. The amount of data to be logged is minimized by combining modified regions of an object if possible.

PRISMA/DB[1] is a main memory relational database with a design that emphasizes use of parallelism to achieve high performance query processing. By running multiple execution sites on each VoltDB node, we were able to utilize the parallelism offered by multiple cores and achieved high performance numbers on our two benchmarks (detailed numbers can be found in Chapter 5).

Purely logical logging has also been proposed recently [22]. Our work in this thesis applies this idea in its extreme to an in-memory database and shows results comparing highly logical command logging to a more traditional logging approach like ARIES.

Recent work by Cao et al [3] describes main-memory checkpoint recovery algorithms for a specific class of OLTP applications called frequently consistent applications. Related work such as [14][29][30] 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 and we deal with all OLTP workloads, so these techniques do not apply.

Chapter 8

Conclusion

In this thesis, we compared the performance of command logging to ARIES at run-time and for recovery in high-throughput OLTP settings. Command logging recovers by re-running committed transactions from a transactionally-consistent checkpoint, whereas ARIES recovers by recording fine-grained updates and re-applying those updates 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), ARIES 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 [11]).

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 the equivalent serial order of execution of the committed transactions in the database.

Bibliography

[1] Peter M. G. Apers, Care A. Van Den Berg, Jan Flokstra, Paul W. P. J. Grefen, Martin L. Kersten, and Annita N. Wilschut. Prisma/db: A parallel main memory relational dbms. *IEEE Transactions on Knowledge and Data Engineering*, 4:541–554, 1992.

[2] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, December 2009.

[3] Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 265–276, New York, NY, USA, 2011. ACM.

[4] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM.

[5] Margaret H. Dunham, Le Gruenwald, Margaret H Dunham, Jing Huang, Jun lin Lin, and Ashley Chaffin Peltier. Recovery in main memory databases, 1996.

[6] Margaret H. Eich. Main memory database recovery. In *Proceedings of 1986 ACM Fall joint computer conference*, ACM '86, pages 1226–1232, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[7] H. Garcia-Molina, R.J. Lipton, and J. Valdes. A massive memory machine. *Computers, IEEE Transactions on*, C-33(5):391 –399, may 1984.

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

[9] Vibby Gottemukkala and Tobin J. Lehman. Locking and latching in a memory-resident database system. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 533–544, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[10] Le Gruenwald and Margaret H. Eich. MMDB reload algorithms. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, SIGMOD '91, pages 397–405, New York, NY, USA, 1991. ACM.

[11] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM.

[12] H. V. Jagadish, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, and S. Sudarshan. Dalí: A high performance main memory storage manager. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 48–59. Morgan Kaufmann, 1994.

[13] H. V. Jagadish, Abraham Silberschatz, and S. Sudarshan. Recovering from main-memory lapses. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 391–404, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[14] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: a scalable approach to logging. *Proc. VLDB Endow.*, 3:681–692, September 2010.

[15] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 603–614, New York, NY, USA, 2010. ACM.

[16] T.J. Lehman, E.J. Shekita, and L.-F. Cabrera. An evaluation of starburst's memory resident storage component. *Knowledge and Data Engineering, IEEE Transactions on*, 4(6):555 –566, December 1992.

[17] Tobin J. Lehman and Michael J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, SIGMOD '87, pages 104–117, New York, NY, USA, 1987. ACM.

[18] Tobin J. Lehman and Michael J. Carey. A concurrency control algorithm for memory-resident database systems. In *Proceedings of the 3rd International Conference on Foundations of Data Organization and Algorithms*, FOFO '89, pages 490–504, London, UK, 1989. Springer-Verlag.

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

[20] Xi Li and Margaret H. Eich. Post-crash log processing for fuzzy checkpointing main memory databases. In *Proceedings of the Ninth International Conference on*

Data Engineering, pages 117–124, Washington, DC, USA, 1993. IEEE Computer Society.

[21] Jun lin Lin and Margaret H. Dunham. Segmented fuzzy checkpointing for main memory databases. In *In Proceedings of the 11th Annual Symposium on Applied Computing (SAC '96*, pages 158–165, 1996.

[22] David Lomet, Kostas Tzoumas, and Michael Zwilling. Implementing performance competitive logical recovery. *Proc. VLDB Endow.*, 4:430–439, April 2011.

[23] C. Mohan. Aries/kvl: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 392–405, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[24] C. Mohan. Aries/lhs: A concurrency control and recovery method using write-ahead logging for linear hashing with separators. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 243–252, Washington, DC, USA, 1993. IEEE Computer Society.

[25] C. Mohan. Repeating history beyond aries. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 1–17, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[26] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter 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.

[27] C. Mohan and Frank Levine. Aries/im: an efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, pages 371–380, New York, NY, USA, 1992. ACM.

[28] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. In *SIGOPS OSR*. Stanford InfoLab, 2009.

[29] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3:928–939, September 2010.

[30] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. Plp: page latch-free shared-everything oltp. *Proc. VLDB Endow.*, 4:610–621, July 2011.

[31] Andrew Pavlo, Carlo Curino, and Zdonik Stan. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. SIGMOD '12, 2012.

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

[33] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[34] Redis. `http://redis.io`.

[35] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

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

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

[38] Seth J. White and David J. DeWitt. Quickstore: a high performance mapped object store. *The VLDB Journal*, 4:629–673, October 1995.