Adaptive Logging: Optimizing Logging and Recovery Costs in Distributed In-memory Databases

Chang Yao[‡] Divyakant Agrawal[♯] Gang Chen[§] Beng Chin Ooi[‡] Sai Wu[§]
[‡]National University of Singapore [♯]University of California at Santa Barbara [§]Zhejiang University
[‡]{yaochang,ooibc}@comp.nus.edu.sg [♯]agrawal@cs.ucsb.edu [§]{cg,wusai}@zju.edu.cn

ABSTRACT

By maintaining the data in main memory, in-memory databases dramatically reduce the I/O cost of transaction processing. However, for recovery purposes, in-memory systems still need to flush the log to disk, which incurs a substantial number of I/Os. Recently, command logging has been proposed to replace the traditional data log (e.g., ARIES logging) in in-memory databases. Instead of recording how the tuples are updated, command logging only tracks the transactions that are being executed, thereby effectively reducing the size of the log and improving the performance. However, when a failure occurs, all the transactions in the log after the last checkpoint must be redone sequentially and this significantly increases the cost of recovery.

In this paper, we first extend the command logging technique to a distributed system, where all the nodes can perform their recovery in parallel. We show that in a distributed system, the only bottleneck of recovery caused by command logging is the synchronization process that attempts to resolve the data dependency among the transactions. We then propose an adaptive logging approach by combining data logging and command logging. The percentage of data logging versus command logging becomes a tuning knob between the performance of transaction processing and recovery to meet different OLTP requirements, and a model is proposed to guide such tuning. Our experimental study compares the performance of our proposed adaptive logging, ARIES-style data logging and command logging on top of H-Store. The results show that adaptive logging can achieve a 10x boost for recovery and a transaction throughput that is comparable to that of command logging.

1. INTRODUCTION

While in-memory databases have been studied since the 80s, recent advances in hardware technology have re-generated interest in hosting the entirety of the database in memory in order to provide faster OLTP and real-time analytics. Simply replacing the storage layer of a traditional disk-based database with memory does not satisfy the real-time performance requirements because of the retention of the complex components from traditional database systems, such as the buffer manager, latching, locking and logging [12, 38, 42]. This calls for re-examination and re-design of many core system components [20, 41]. In this paper, we focus on logging and recovery efficiency in distributed in-memory databases.

SIGMOD'16 June 26 - July 01, 2016, San Francisco, CA, USA
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-3531-7/16/06.
DOI: http://dx.doi.org/10.1145/2882903.2915208

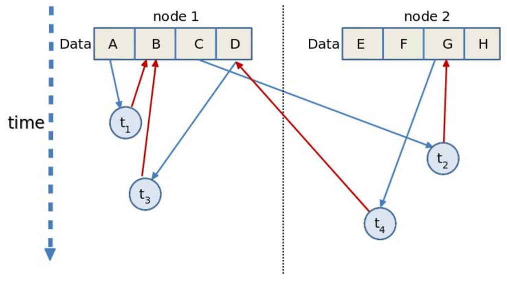


Figure 1: Example of logging techniques

Recovery is an indispensable and expensive process of database systems that ensures updates made by committed transactions before any system crash are reflected in the database after recovery, while updates made by uncommitted transactions are ignored. In an OLTP system, logging for the purpose of recovery affects the system throughput, and recovery affects the system availability and hence its overall throughput. Therefore, efficient logging and recovery are important to the overall performance of an OLTP system. While node failures are infrequent, an increasingly large number of compute nodes in a distributed system inevitably leads to an increase in the probability of node failures [30, 32, 36]. Consequently, increasing number of failed nodes results in higher failure costs as more recoveries are needed. Therefore, providing an effective failure recovery strategy is as imperative as efficient logging for distributed database systems.

The most widely used logging method in conventional databases is the write-ahead log (e.g., ARIES log [23]). The idea is to record how tuples are being updated by the transactions, and we refer to it as the data log in our discussion. Let us consider Figure 1 as an example. There are two nodes processing four concurrent transactions, t_1 to t_4. All the transactions follow the same format:

$$f(x, y) : y = 2x$$

So $t_1 = f(A, B)$, indicating that t_1 reads the value of A and then updates B as $2A$. Since different transactions may modify the same value, a locking mechanism is required. Based on their timestamps, the correct serialized order of the transactions is t_1, t_2, t_3 and t_4. Let $v(X)$ denote the value of parameter X. The ARIES data log of the four transactions are listed as in Table 1.

ARIES log records how the data are modified by the transactions, and supports efficient recovery using the log data when a failure occurs. However, the recovery process of in-memory databases is

Table 1: ARIES log

timestamp	transaction ID	parameter	old value	new value
100001	t_1	B	v(B)	2v(A)
100002	t_2	G	v(G)	2v(C)
100003	t_3	B	v(B)	2v(D)
100004	t_4	D	v(D)	2v(G)

sightly different from that of their disk-based analog. It first loads the database snapshot recorded in the last checkpoint and then replays all the committed transactions in ARIES log. For uncommitted transactions, no roll-backs are required as the corresponding updates are not persistent onto disk yet.

ARIES log is a "heavy-weight" logging method, as it incurs high overhead. In conventional databases, where I/Os for processing transactions dominate the performance, the logging overhead is a small fraction of the overall cost. However, in an in-memory database where all the transactions are processed in memory, logging cost dominates the overall cost.

To reduce the logging overhead, a command logging approach [22] is proposed to only record the transaction information with which transaction can be fully replayed during failure recovery. In H-Store [16], each command log records the ID of the corresponding transaction and the stored procedure that has been applied to update the database along with the input parameters. As an example, the command logging records for Figure 1 are simplified in Table 2.

Table 2: Command log

transaction ID	timestamp	stored procedure pointer	parameters
1	100001	p	A, B
2	100002	p	C, G
3	100003	p	D, B
4	100004	p	G, D

Since all four transactions follow the same routine, we only keep a pointer p to the details of stored procedure: $f(x, y) : y = 2x$. For recovery purposes, we also need to maintain the parameters for each transaction, so that the system can re-execute all the transactions from the last checkpoint when a failure happens. Compared with ARIES-style log, a command log is much more compact and hence reduces the I/O cost for materializing it onto disk. It has been shown that command logging can significantly increase the throughput of transaction processing in in-memory databases [22]. However, the improvement is achieved at the expense of much longer recovery time after a failure since transactions have to be re-executed to recreate the database state.

When there is a node failure, all the transactions have to be replayed in the command logging method, while ARIES-style logging simply recovers the value of each column. Throughout the paper, we use "attribute" to refer a column defined in the schema and "attribute value" to denote the value of a tuple in a specific column. For example, to fully redo transaction t_1, command logging needs to read the value of A and update the value of B, whereas for ARIES logging, we just need to set B's value as $2v(A)$ as recorded in the log file. More importantly, command logging does not support parallel recovery in a distributed system. In command logging [22], command logs of different nodes are merged at the master node during recovery. To guarantee the correctness of recovery, transactions must be reprocessed in a serialized order based on their timestamps. For example, even in a cluster consisting of two nodes, the transactions have to be replayed one by one due to their possible contentions. In the previous example, transaction t_3 and t_4 cannot be concurrently processed by node 1 and 2 respectively, because both transactions need to lock the value of D. There is no such limitation in the ARIES-style logging that actually partitions the log data into different nodes. In Table 1, log records of t_1, t_3 and t_4 are stored in node 1, while the record of t_2 is maintained by node 2. Both nodes can start their recovery process concurrently and independently. All contentions can be resolved locally by each node.

In command logging, a single-node failure will trigger a complete recovery process because we do not know which parts of the updates are lost. For example, if node 2 in Figure 1 fails, both node 1 and node 2 need to roll back and replay transaction t_1 to t_4. Suppose an in-memory database is hosted over hundreds of nodes [1, 3, 26], where failures cannot be assumed as exceptions, command logging could adversely affect the overall system throughput. If ARIES-style log is applied instead, each node maintains all data updates in its local log file. Therefore, we only need to recover the failed node using its own log data and no global recovery is required. In summary, command logging reduces the I/O cost of processing transactions, but incurs a much higher cost for recovery than ARIES-style logging, especially in a distributed environment. To this end, we propose a new logging scheme that achieves a comparable performance as command logging for processing transactions, while enabling a much more efficient recovery. Our logging approach also allows the users to tune the parameters to achieve a preferable trade-off between transaction processing and recovery.

In this paper, we first propose a distributed version of command logging. In the recovery process, before redoing the transactions, we first generate the dependency graph by scanning the log data. Transactions that read or write the same tuple will be linked together. A transaction can be reprocessed only after all its dependent transactions have been committed. On the other hand, the transactions that do not have dependency relationship can be concurrently processed. Based on this principle, we organize the transactions into different processing groups. Transactions inside a group have dependency relationship, while transactions of different groups can be processed concurrently.

While the distributed version of command logging effectively exploits the parallelism among the nodes to speed up recovery, some processing groups could be rather large, causing a few transactions to block the processing of many others. We subsequently propose an adaptive logging approach that adaptively makes use of the command logging and ARIES-style logging. Specifically, we identify the bottlenecks dynamically based on a cost model and resolve them using ARIES logging. We materialize the transactions identified as bottlenecks in ARIES log so that transactions depending on them can be recovered more efficiently. In an extreme case, if all the transactions are identified as bottlenecks, our method spends the same I/O cost as the ARIES logging. However, this will definitely increase the cost of transaction processing. Therefore, our adaptive logging method selectively creates ARIES logs using a cost model based on a given budget. The budget is used as a parameter in adaptive logging for users to tune the trade-off between transaction processing and recovery.

It is indeed very challenging to classify the transactions into the ones that may cause bottleneck and those that will not, because we have to make a real-time decision on adopting either command logging or ARIES logging. During transaction processing, we do not know the impending distribution of transactions, and even if the dependency graph of impending transactions is known beforehand, the optimization problem of log creation remains to be NP-hard. Hence, we propose both offline and online heuristic approaches to derive approximate solutions, by estimating the importance of each transaction based on the access patterns of existing transactions. The offline approach provides the ideal logging plan and is used mainly for comparison in our performance study, while our system uses the online approach.

Finally, we implement our two proposed methods, namely distributed command logging and adaptive logging, on top of H-Store [16] and compare them with ARIES logging and command logging. Experimental results show that adaptive logging can achieve a comparable performance for transaction processing as command logging, while it performs 10x faster than command logging for recovery in a distributed system.

The remainder of the paper is organized as follows. We present our distributed command logging method in Section 2 and the new adaptive logging method in Section 3. The experimental results are presented in Section 4 and we review some related work in Section 5. The paper is concluded in Section 6.

2. DISTRIBUTED COMMAND LOGGING

As described earlier, command logging only records the transaction ID, stored procedure and its input parameters [22]. Once failure occurs, the database can restore the last snapshot and redo all the transactions in the command log to re-establish the database state. Command logging operates at a much coarser granularity and writes much fewer bytes per transaction than ARIES-style logging. However, the major concern of command logging is its recovery performance.

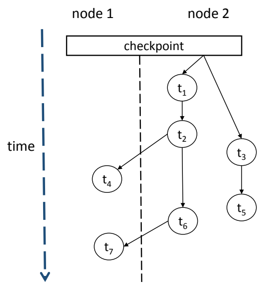
In H-Store [16] and its commercial successor, VoltDB[1], command logs of different nodes are shuffled to the master node that merges them based on the timestamp order. Since command logging does not record how the data are manipulated in each page, all the transactions must be redone sequentially, incurring a high recovery overhead. An alternative solution is to maintain multiple replicas, so that data on the failed nodes can be recovered from other replicas. However, the drawback of such an approach is twofold. First, maintaining the consistency among replicas incurs a high synchronization overhead and as a consequence slows down the transaction processing. Second, given a limited amount of memory, it is too expensive to maintain replicas in memory, especially when processing large scale dataset. This is confirmed by a performance study of a replication-based technique in Appendix C. In this paper, we shall therefore focus on the log-based approaches.

Even with command logging, it is unnecessary to replay all committed transactions by examining the correlations among transactions. Our intuition is to design a distributed command logging method that enables all nodes to start their recovery in parallel and replay only the necessary transactions.

We first define the correctness of recovery in our system. Suppose the data are partitioned to N cluster nodes. Let $\mathcal{T}$ be the set of transactions since the last checkpoint. For a transaction $t_i \in \mathcal{T}$, if t_i reads or writes a tuple on node $n_x \in N$, n_x is marked as a participant of t_i. Specifically, we use $f(t_i)$ to denote all those nodes involved in t_i and use $f^{-1}(n_x)$ to represent all the transactions in which n_x has participated. In a distributed system, each transaction has a coordinator, typically the node that minimizes the data transfer for processing the transaction. The coordinator schedules the data accesses and monitors how its transaction is processed. Hence, the command log entry is usually created in the coordinator [22]. We use $\theta(t_i)$ to denote t_i's coordinator. Obviously, we have $\theta(t_i) \in f(t_i)$.

Given two transactions $t_i, t_j \in \mathcal{T}$, we define an order function $\prec$ as: $t_i \prec t_j$, only if t_i is committed before t_j. When a node n_x fails, a set of transactions $f^{-1}(n_x)$ need to be redone. But these transactions may compete for the same tuple with other transactions. Let $s(t_i)$ and $c(t_i)$ denote the submission time and commit time of a transaction t_i respectively.

[1]http://voltdb.com/



txn	submission time	commit time	stored procedure	parameters
1	1	2	$f_1(p_1)$: $p_1=p_1+1$	x_1
2	2	4	$f_2(p_1,p_2)$: $p_1=p_1-1$, $p_2=p_1+2$	x_1,x_3
3	3	4	$f_3(p_1)$: $p_1=2*p_1$	x_4
4	4	5	$f_4(p_1)$: $p_1=\log(p_1)$	x_3
5	4	5	$f_5(p_1, p_2)$: $p_1=2*p_2$	x_4, x_5
6	5	6	$f_5(p_1, p_2)$: $p_1=2*p_2$	x_6, x_1
7	6	7	$f_1(p_1)$: $p_1=p_1+1$	x_6

Figure 2: A running example

DEFINITION 1. **Transaction Competition**
Transaction t_i competes with transaction t_j, if the two conditions below are satisfied.

1. $s(t_j) < s(t_i) < c(t_j)$.
2. *t_i and t_j read or write the same tuple.*

Accordingly, transaction competition is defined as a unidirectional relationship: t_i competes with t_j, and t_j may compete with the others that may modify the same set of tuples and commit before it. We further let $\odot(t_i)$ be the set of all the transactions that t_i competes with, and define function g for transaction set $\mathcal{T}_j$ as:

$$g(\mathcal{T}_j) = \bigcup_{\forall t_i \in \mathcal{T}_j} \odot(t_i)$$

To recover the database from n_x's failure, an initial recovery set $\mathcal{T}_0^x = f^{-1}(n_x)$ is created and set $\mathcal{T}_{i+1}^x = \mathcal{T}_i^x \cup g(\mathcal{T}_i^x)$. As the number of committed transactions is limited, there exists a minimum number L such that when $j \geq L$, $\mathcal{T}_{j+1}^x = \mathcal{T}_j^x$, and there are no other transactions accessing the same set of tuples since the last checkpoint. We call $\mathcal{T}_L^x$ the complete recovery set for n_x.

Next, we define the correctness of recovery in the distributed system as:

DEFINITION 2. **Correctness of Recovery**
When node n_x fails, all the transactions in its complete recovery set need to be redone by strictly following their commit order, i.e., if $t_i \prec t_j$, then t_i must be reprocessed before t_j.

To recover from a node's failure, our approach needs to retrieve its complete recovery set. For this purpose, it builds a dependency graph.

2.1 Dependency Graph

Dependency graph is defined as a directed acyclic graph $G = (V, E)$, where each vertex in V is a transaction t_i, containing the information about its timestamps ($c(t_i)$ and $s(t_i)$) and coordinator $\theta(t_i)$. t_i has an edge to t_j, if and only if

1. $t_i \in \odot(t_j)$
2. $\forall t_m \in \odot(t_j), c(t_m) < c(t_i)$

For a specific order of transaction processing, there is one unique dependency graph as shown in the following theorem.

THEOREM 1. *Given a transaction set $\mathcal{T} = \{t_0, ..., t_k\}$, where $c(t_i) < c(t_{i+1})$, there exists a unique dependency graph for $\mathcal{T}$.*

PROOF. Since the vertices represent transactions, we always have the same set of vertices for the same set of transactions. We only need to prove that the edges are also unique. Based on the definition, edge e_{ij} exists, only if there exists one set of tuples that are accessed by t_j and updated by t_i and no other transactions that have been committed between t_i and t_j would update the same set of tuples. Therefore, edge e_{ij} is a unique edge between t_i and t_j. □

A running example that illustrates the idea is shown in Figure 2. There are seven transactions since the last checkpoint, aligned based on their coordinators: node 1 and node 2. The transaction IDs, submission times, commit times, stored procedures and parameters are also shown in the table. Based on the definition, transaction t_2 competes with transaction t_1, as both of them update x_1. Transaction t_4 competes with transaction t_2 on x_3. The complete recovery set for t_4 is $\{t_1, t_2, t_4\}$. Note that although t_4 does not access the attribute that t_1 updates, t_1 is still in t_4's recovery set because of the transitive dependency among t_1, t_2 and t_4. After having constructed the dependency graph and generating the recovery set, the failed node can then be recovered adaptively. For example, to recover node 1, we do not have to reprocess transactions t_3 and t_5.

2.2 Processing Group

In order to generate the complete recovery set efficiently, we organize transactions as processing groups. Algorithm 1 and 2 illustrate how the groups are generated from a dependency graph. In Algorithm 1, we start from the root vertex that represents the checkpoint to iterate over all the vertices in the graph. The neighbors of the root vertex are transactions that do not compete with the others. We create one processing group for each of them (line 3-7). *AddGroup* is a recursive function that explores all reachable vertices and adds them into the group. One transaction may exist in multiple groups if more than one transactions compete with it.

Two indexes are built for each processing group. One is an inverted index that supports efficient retrieval of the participating groups of a given transaction. The other index creates a backward link from each transaction to the transactions it competes with. Using these two indexes, it is efficient to generate the complete recovery set for a specific transaction.

Algorithm 1 `CreateGroup(DependencyGraph` G`)`

```
1: Set $S = \emptyset$
2: Vertex $v$ = $G$.getRoot()
3: **while** $v$.hasMoreEdge() **do**
4:     Vertex $v_0$ = $v$.getEdge().endVertex()
5:     Group $g$ = new Group()
6:     AddGroup($g$, $v_0$)
7:     $S$.add($g$)
8: return $S$
```

2.3 Algorithm for Distributed Command Logging

When a failure on node n_x is detected, the system stops the transaction processing and starts the recovery process. One new node is started to reprocess all the transactions in n_x's complete recovery set. Since some transactions are distributed transactions involving multiple nodes, the recovery algorithm runs as a distributed process.

Algorithm 3 shows the basic idea of recovery. First, it retrieves all the transactions that do not compete with the others since the

Algorithm 2 `AddGroup(Group` g`, Vertex` v`)`

```
1: $g$.add($v$)
2: **while** $v$.hasMoreEdge() **do**
3:     Vertex $v_0$ = $v$.getEdge().endVertex()
4:     AddGroup($g$, $v_0$)
```

last checkpoint (line 3). These transactions can be processed in parallel. Therefore, they would be forwarded to the corresponding coordinators for processing. At each coordinator, Algorithm 4 is invoked to process a specific transaction t. t first waits until all the transactions in $\odot(t)$ are processed. Then it would be processed and all its neighbor transactions are retrieved by following the links in the dependency graph. If t's neighbor transactions are also in the recovery set, *ParallelRecovery* function would be invoked recursively to process them.

THEOREM 2. *Distributed command logging algorithm guarantees the correctness of the recovery.*

PROOF. In Algorithm 3, if two transactions t_i and t_j are in the same processing group and $c(t_i) < c(t_j)$, t_i must be processed before t_j, as we follow the links of dependency graph. The complete recovery set of t_j is the subset of the union of all the processing groups that t_j joins. Therefore, we will redo all the transactions in the recovery set for a specific transaction as in Algorithm 3. □

As an example, suppose node 1 fails in Figure 2. The recovery set is $\{t_1, t_2, t_4, t_6, t_7\}$. t_1 on node 2 is the only transaction that can run without waiting for the other transactions. It will be redone first. Note that although node 2 does not fail, t_1 still needs to be reprocessed, because it affects the correctness when reprocessing the committed transactions on node 1. After t_1 and t_2 commit, the new node that replaces node 1 will reprocess t_4. Simultaneously, node 2 will process t_6 in order to recover t_7. The example shows that compared with the original command logging method, our distributed command logging has the following two advantages:

1. Only the involved transactions need to be reprocessed. Therefore, the recovery processing cost is reduced.

2. Different processing nodes can perform the recovery process concurrently, leveraging the parallelism to speed up the recovery.

Algorithm 3 `Recover(Node` n_x`, DependencyGraph` G`)`

```
1: Set $S_T$ = getAllTransactions($n_x$)
2: CompleteRecoverySet $S$=getRecoverySet($G$,$S_T$)
3: Set $S_R$ = getRootTransactions($S$)
4: **for** Transaction $t \in S_R$ **do**
5:     Node $n$ = $t$.getCoordinator()
6:     ParallelRecovery($n$, $S_T$, $t$)
```

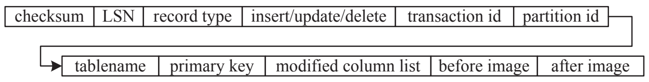
2.4 Footprint of Transactions

To reduce the overhead of transaction processing, a dependency graph is built offline. Before a recovery process starts, the dependency graph is built by scanning the logs. For this purpose, we introduce a light weight write ahead footprint log for the transactions. Once a transaction is committed, we record the transaction ID and the involved tuple IDs as its footprint log.

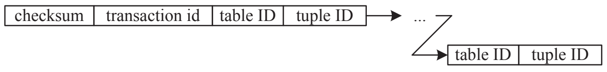
Figure 3 illustrates the structure of footprint log and ARIES log. ARIES log maintains the detailed information about a transaction, including partition ID, table name, modified column, original value

Algorithm 4 `ParallelRecovery(Node` n_x`, Set` S_T`, Transaction` t`)`

```
1: while wait($\odot(t)$) do
2:     sleep(timethreshold)
3: if $t$ has not been processed then
4:     process($t$)
5:     Set $S_t$ = $g$.getDescendant($t$)
6:     for $\forall t_i \in S_t \cap S_T$ do
7:         Node $n_i$ = $t$.getCoordinator()
8:         ParallelRecovery($n_i$, $S_T$, $t_i$)
```



(a) ARIES Log



(b) Footprint Log

Figure 3: ARIES log versus footprint log

and updated value, based on which we can correctly redo a transaction. On the contrary, a footprint log only records IDs of those tuples that are read or updated by a transaction. It consumes less storage space than ARIES log (on average, each record in ARIES log and footprint log require 3KB and 450B respectively on the TPC-C benchmark). The objective of maintaining footprint log is not for recovering the lost transactions, but for building the dependency graph.

3. ADAPTIVE LOGGING

The bottleneck of distributed command logging is caused by the dependencies among the transactions. To ensure consistency [5], transaction t is blocked until all the transactions in $\odot(t)$ have been processed. In an ideal case, where no transactions compete for the same tuple, the maximal parallelism can be achieved by recovering all transactions in parallel. Otherwise, transactions that compete with each other must be synchronized, which significantly slows down the recovery process. If the dependencies among the transactions are fully or partially resolved, the cost of recovery could be effectively reduced.

3.1 The Key Concept

As noted in the introduction, ARIES log allows each node to recover independently. If node n_x fails, all the updates since the last checkpoint would be redone by scanning its log. Recovery with ARIES log does not need to take transaction dependency into account, because the log completely records how a transaction modifies the data. Hence, the intuition of our adaptive logging approach is to combine command logging and ARIES logging. For transactions that are highly dependent on the others, ARIES logs are created to speed up their reprocessing during the recovery. For other transactions, command logging is applied to reduce the logging overhead at runtime.

For the example shown in Figure 2, if ARIES log has been created for t_7, there is no need to reprocess t_6 during the recovery of node 1. Moreover, if ARIES log has been created for t_2, the recovery process just needs to redo t_2 and then t_4 rather than starting from t_1. For the recovery of node 1, only three transactions, namely $\{t_2, t_4, t_7\}$, need to be re-executed. To determine whether a transaction depends on the results of other transactions, we need a new relationship in addition to the transaction competition to describe the causal consistency [5].

DEFINITION 3. **Time-Dependent Transaction**
Given two transaction t_i and t_j, t_j is t_i's time-dependent transaction, if and only if

1. *$c(t_i) > c(t_j)$*
2. *t_j updates an attribute a_x of tuple r that is accessed by t_i*
3. *there is no other transaction with commit time between $c(t_i)$ and $c(t_j)$ that also updates $r.a_x$*

Let $\otimes(t_i)$ denote all the time-dependent transactions of t_i. For transactions in $\otimes(t_i)$, their time-dependent transactions can also be found recursively, denoted as $\otimes^2(t_i) = \otimes(\otimes(t_i))$. This process continues until it finds the minimal x satisfying $\otimes^x(t_i) = \otimes^{x+1}(t_i)$. $\otimes^x(t_i)$ represents all the transactions that must run before t_i to guarantee the causal consistency. In a special case, if transaction t_i does not compete with the others, it does not have time-dependent transactions (namely, $\otimes(t_i) = \emptyset$) either. $\otimes^x(t_i)$ is a subset of the complete recovery set of t_i. Instead of redoing all the transactions in the complete recovery set, t_i can be recovered correctly by only reprocessing the transactions in $\otimes^x(t_i)$.

If some transactions in $\otimes^x(t_i)$ are adaptively selected to create ARIES logs, the recovery cost of t_i can be effectively reduced. In other words, if ARIES log is created for transaction $t_j \in \otimes^x(t_i)$, $\otimes(t_j) = \emptyset$ and $\otimes^x(t_j) = \emptyset$, because t_j can now be recovered by simply loading its ARIES log since it does not depend on the results of the other transactions.

More specifically, let $A = \{a_0, a_1, ..., a_m\}$ denote the attributes that t_i needs to access. These attributes may come from different tuples. Let $\otimes(t_i.a_x)$ represent the time-dependent transactions of t_i that have updated a_x. Therefore, $\otimes(t_i) = \otimes(t_i.a_0) \cup ... \cup \otimes(t_i.a_m)$. To formalize the usage of ARIES log in reducing the recovery cost, we introduce the following lemmas.

LEMMA 1. *If an ARIES log has been created for $t_j \in \otimes(t_i)$, transactions t_l in $\otimes^{x-1}(t_j)$ can be discarded from $\otimes^x(t_i)$, if*

$$\nexists t_m \in \otimes(t_i), t_m = t_l \vee t_l \in \otimes^{x-1}(t_m)$$

PROOF. The lemma states that all the time-dependent transactions of t_j can be discarded, if they are not time-dependent transactions of the other transactions in $\otimes(t_i)$, which is obviously true. □

The above lemma can be further extended for an arbitrary transaction in $\otimes^x(t_i)$.

LEMMA 2. *Suppose an ARIES log has been created for transaction $t_j \in \otimes^x(t_i)$ that updates attribute set $\bar{A}$. Transaction $t_l \in \otimes^x(t_j)$ can be discarded, if*

$$\nexists a_x \in (A - \bar{A}), t_l \in \otimes^x(t_i.a_x)$$

PROOF. Since t_j updates $\bar{A}$, all the transactions in $\otimes^x(t_j)$ that only update attribute values in $\bar{A}$ can be discarded without violating the correctness of causal consistency. □

Lemma 2 shows that the time-dependent transactions of t_j are not necessary in the recovery process, if they are not time-dependent transactions of any attribute in $(A - \bar{A})$. Recovery of the values of attribute set $\bar{A}$ for t_i can start from t_j's ARIES log by redoing t_j, and then all the transactions with timestamps falling in the range $(c(t_j), c(t_i))$ that also update $\bar{A}$. To simplify the presentation for the time being, we use $\phi(t_j, t_i, t_j.\bar{A})$ to denote these transactions.

Finally, we summarize our observations as the following theorem, based on which we design our adaptive logging and the corresponding recovery algorithm.

THEOREM 3. *Suppose* ARIES *logs have been created for transaction set $\mathcal{T}_a$, to recover t_i, it is sufficient to redo all the transactions in the following set.*

$$\bigcup_{\forall a_x \in (A - \bigcup_{\forall t_j \in \mathcal{T}_a} t_j.\bar{A})} \otimes^x(t_i.a_x) \cup \bigcup_{\forall t_j \in \mathcal{T}_a} \phi(t_j, t_i, t_j.\bar{A})$$

PROOF. t_i can be recovered correctly by redoing all its time-dependent transactions from the latest checkpoint. Based on Lemma 1 and Lemma 2, the number of redoing transactions can be reduced by creating ARIES logs. The first term of the formula represents all the transactions that are required to recover attribute values in $(A - \bigcup_{\forall t_j \in \mathcal{T}_a} t_j.\bar{A})$, while the second term denotes all those transactions that we need to recover from ARIES logs following the timestamp order. □

3.2 Logging Strategy

By combining ARIES logging and command logging into a hybrid logging approach, the recovery cost can be effectively reduced. Users should first specify an I/O budget $B_{i/o}$ according to their requirements. With a given I/O budget $B_{i/o}$, our adaptive approach selects the transactions for ARIES logging to maximize the recovery performance. This decision of which type of logs to create for each transaction has to be made during transaction processing. However, since the distribution future of transactions cannot be known in advance, it is impossible to generate an optimal selection. In fact, even if we know all the future transactions, the optimization problem is still NP-hard.

Let $w^{aries}(t_j)$ and $w^{cmd}(t_j)$ denote the I/O costs of ARIES logging and command logging for transaction t_j respectively. We use $r^{aries}(t_j)$ and $r^{cmd}(t_j)$ to represent the recovery cost of t_j regarding to the ARIES logging and command logging respectively. If an ARIES log is created for transaction t_j that is a time-dependent transaction of t_i, the recovery cost is reduced by:

$$\Delta(t_j, t_i) = \sum_{\forall t_x \in \otimes^x(t_i)} r^{cmd}(t_x) - \sum_{\forall t_x \in \phi(t_j, t_i, t_j.\bar{A})} r^{cmd}(t_x) - r^{aries}(t_j) \quad (1)$$

If ARIES logs are created for more than one transactions in $\otimes^x(t_i)$, $\Delta(t_j, t_i)$ should be updated accordingly. Let $\mathcal{T}_a \subset \otimes^x(t_i)$ be the set of transactions with ARIES logs. We define an attribute set:

$$p(\mathcal{T}_a, t_j) = \bigcup_{\forall t_x \in \mathcal{T} \wedge c(t_x) > c(t_j)} t_x.\bar{A}$$

$p(\mathcal{T}_a, t_j)$ represent the attributes that are updated after t_j by the transactions with ARIES logs. Therefore, $\Delta(t_j, t_i)$ is adjusted as

$$\Delta(t_j, t_i, \mathcal{T}_a) = \sum_{\forall t_x \in \otimes^x(t_i) - \mathcal{T}_a} r^{cmd}(t_x) - r^{aries}(t_j) - \sum_{\forall t_x \in \phi(t_j, t_i, t_j.\bar{A} - p(\mathcal{T}_a, t_j))} r^{cmd}(t_x) \quad (2)$$

DEFINITION 4. **Optimization Problem of Logging Strategy** *For transaction t_i, find a transaction set $\mathcal{T}_a$ to create* ARIES *logs so that:*

$$\textit{maximize} \sum_{\forall t_j \in \mathcal{T}_a} \Delta(t_j, t_i, \mathcal{T}_a) \textit{ s.t.} \sum_{\forall t_j \in \mathcal{T}_a} w^{aries}(t_j) \le B_{i/o}$$

We note that the single transaction optimization problem is analogous to the 0-1 knapsack problem, while the more general case is similar to the multi-objective knapsack problem. It becomes more complicated when function Δ is also determined by the correlations of transactions. We therefore design both offline and online heuristic methods to derive approximate solutions.

3.2.1 Offline Algorithm

We first introduce our offline algorithm designed for the ideal case, where the impending distribution of transactions is known. The offline algorithm is only used to demonstrate the basic idea of adaptive logging, while our system employs its online variant. $\mathcal{T}$ is used to represent all the transactions from the last checkpoint to the point of failure.

For each transaction $t_i \in \mathcal{T}$, the benefit of creating ARIES log is computed as:

$$b(t_i) = \sum_{\forall t_j \in \mathcal{T} \wedge c(t_i) < c(t_j)} \Delta(t_i, t_j, \mathcal{T}_a) \times \frac{1}{w^{aries}(t_i)}$$

Initially, $\mathcal{T}_a = \emptyset$.

The transactions are sorted based on their benefit values. The one with the maximal benefit is selected and added to $\mathcal{T}_a$. All the transactions update their benefits accordingly based on Equation 2. This process continues until

$$\sum_{\forall t_j \in \mathcal{T}_a} w^{aries}(t_j) \le B_{i/o}$$

Algorithm 5 outlines the basic idea of the offline algorithm.

Algorithm 5 `Offline(TransactionSet` $\mathcal{T}$`)`

```
1: Set $\mathcal{T}_a = \emptyset$, Map benefits;
2: **for** $\forall t_i \in \mathcal{T}$ **do**
3:     benefits[$t_i$] = computeBenefit($t_i$)
4: **while** getTotalCost($\mathcal{T}_a$)< $B_{i/o}$ **do**
5:     sort(benefits)
6:     $\mathcal{T}_a$.add(benefits.keys().first())
7: return $\mathcal{T}_a$
```

Since Algorithm 5 needs to re-sort all the transactions after each update to $\mathcal{T}_a$, the complexity of the algorithm is $O(l^2)$, where l is the number of transactions. In practice, full sorting can be avoided in most of the cases, because $\Delta(t_i, t_j, \mathcal{T}_a)$ should be recalculated only if both t_i and t_j update a value of the same attribute.

3.2.2 Online Algorithm

Our online algorithm is similar to the offline algorithm, except that it must choose either ARIES logging or command logging in real-time. Since the distribution of future transactions is unknown, we use a histogram to approximate the distribution. In particular, for all the attributes $A = (a_0, ..., a_k)$ involved in transactions, the number of transactions that read or write a specific attribute value is recorded. The histogram is used to estimate the probability of accessing an attribute a_i, denoted as $P(a_i)$. Note that attributes in A may come from the same tuple or different tuples. For tuple v_0 and v_1, if both $v_0.a_i$ and $v_1.a_i$ appear in A, they will be recorded as two different attributes.

In the previous discussion, we use $\phi(t_j, t_i, t_j.\bar{A})$ to denote the transactions that commit between t_j and t_i and also update some attributes in $t_j.\bar{A}$. We are now ready to rewrite it as:

$$\phi(t_j, t_i, t_j.\bar{A}) = \bigcup_{\forall a_i \in t_j.\bar{A}} \phi(t_j, t_i, a_i)$$

Similarly, let $S = t_j.\bar{A} - p(\mathcal{T}_a, t_j)$. The third term in Equation 2 can be computed as:

$$\sum_{\forall t_x \in \phi(t_j, t_i, S)} r^{cmd}(t_x) = \sum_{\forall a_x \in S} \Big(\sum_{\forall t_x \in \phi(t_j, t_i, a_x)} r^{cmd}(t_x) \Big)$$

During the processing of transaction t_j, the number of the future transactions that are affected by the results of t_j and the cost of recovering a future transaction using command log are both unknown. Therefore, we use a constant R^{cmd} to denote the average recovery cost of command logging. Then the above Equation can be simplified as:

$$\sum_{\forall t_x \in \phi(t_j, t_i, S)} r^{cmd}(t_x) = \sum_{\forall a_x \in S} (P(a_x) R^{cmd}) \qquad (3)$$

The first term in Equation 2 estimates the cost of recovering t_j's time-dependent transactions using command logging. It can be efficiently computed in real-time if we maintain the dependency graph. Therefore, by combining Equation 2 and Equation 3, the benefit $b(t_i)$ of a specific transaction can be estimated during online processing. Supposing ARIES logs already have been created for the transactions in $\mathcal{T}_a$, the benefit should be updated based on Equation 2.

The last problem is how to define a threshold γ to trigger the creation of ARIES log for a transaction when its benefit is greater than the threshold. Let us consider an ideal case. Suppose the node fails while processing t_i and an ARIES log for t_i has been created. This log achieves the maximal benefit that can be estimated as:

$$b_i^{opt} = (\mathbb{N} R^{cmd} \sum_{\forall a_x \in A} P(a_x) - R^{aries}) \times \frac{1}{W^{aries}}$$

where $\mathbb{N}$ denotes the number of transactions committed before t_i, and R^{aries} and W^{aries} are the average recovery cost and I/O cost of an ARIES log respectively.

Suppose the occurrence of failure arbitrarily follows a Poisson distribution with parameter λ. That is, the expected average failure time is λ. Let $\rho(\lambda)$ be the number of committed transactions before the failure. So the possibly maximal benefit is:

$$b^{opt} = (\rho(\lambda) R^{cmd} \sum_{\forall a_x \in A} P(a_x) - R^{aries}) \times \frac{1}{W^{aries}}$$

We define our threshold as $\gamma = \alpha b^{opt}$, where α is a tunable parameter.

Given an I/O budget, approximately $\frac{B_{i/o}}{W^{aries}}$ ARIES log records can be created. As failures may happen randomly at anytime, the log should be evenly distributed over the timeline. More specifically, the cumulative distribution function of the Poisson distribution is

$$P(fail\_time < k) = e^{-\lambda} \sum_{i=0}^{\lfloor k \rfloor} \frac{\lambda^i}{i!}$$

Hence, at the k-th second, we can maximally create

$$quota(k) = P(fail\_time < k) \frac{B_{i/o}}{W^{aries}}$$

ARIES log records. At runtime, the system should check whether it still has the quota for ARIES log. If not, no more ARIES logs will be created.

Finally, the idea of online adaptive logging scheme is summarized in Algorithm 6.

Algorithm 6 `Online(Transaction` t_i`, int` $usedQuota$`)`

```
1: int $q$ = getQuota($s(t_i)$)- $usedQuota$
2: **if** $q > 0$ **then**
3:     Benefit $b$=computeBenefit($t$)
4:     **if** $b > \tau$ **then**
5:         $usedQuota$++
6:         createAriesLog($t_i$)
7:     **else**
8:         createCommandLog($t_i$)
```

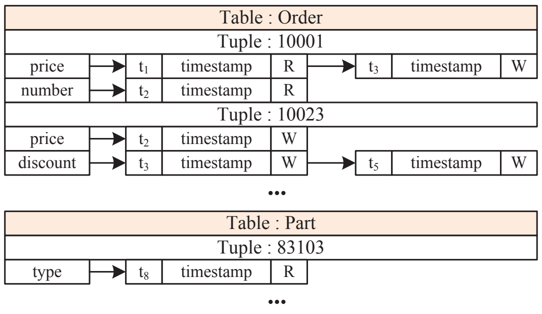


Figure 4: In-memory index

3.3 In-Memory Index

To help compute the benefit of each transaction, an in-memory inverted index is created in our master server. Figure 4 shows the structure of the index. The index entries are organized by table ID and tuple ID. For each specific tuple, we record the transactions that read or write its attributes. As an example, in Figure 4, transaction t_2 reads the *number* of tuple 10001 and updates the *price* of tuple 10023.

By using the index, the time-dependent transactions of a transaction can be efficiently retrieved. For transaction t_5, let A_5 be the attributes that it accesses. The recovery process searches the index to retrieve all the transactions that update any attribute in A_5 before t_5. In Figure 4, because the *discount* value of tuple 10023 is updated by t_5, we check its list and find that t_3 updates the same value before t_5. Therefore, t_3 is a time-dependent transaction of t_5. In fact, the index can also be employed to recover the dependency graph of transactions.

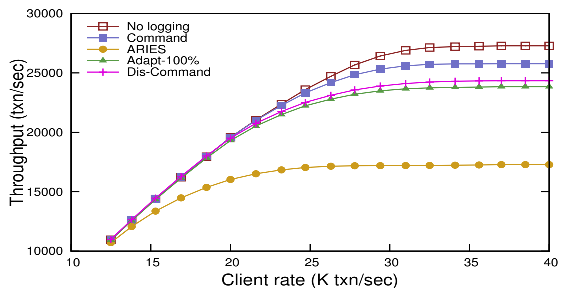
When a transaction is submitted to the system, its information is inserted into the in-memory index. We maintain the index in memory for the following reasons.

1. The index construction should not slow down the transaction processing.

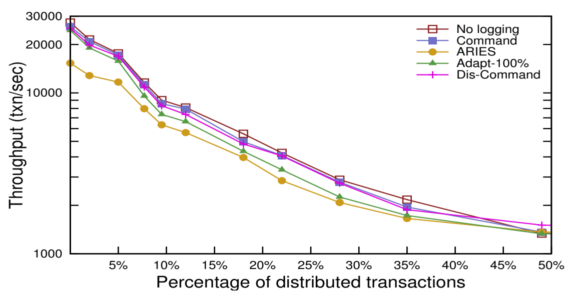
2. The index is small as it only records the necessary information for causal consistency.

3. If the master server fails and the index is lost, system can retain the correctness of transaction processing unaffected, but the performance may be affected. We evaluate such performance impact in Appendix E.1.

Since the in-memory index is used mainly for estimation purpose, it is unnecessary to enforce the index in the master node to be strictly consistent. Therefore, in our current implementation, each processing node caches its in-memory index which is synchronized with that in the master node periodically. We provide more detailed discussion in Appendix D.5.

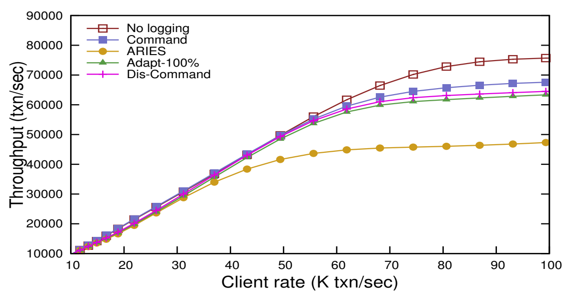


(a) Throughput without distributed transactions

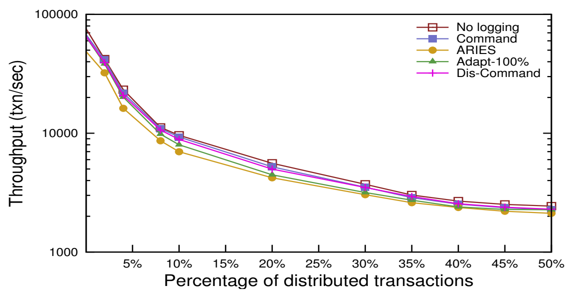


(b) Throughput with distributed transactions (with log scale on y-axis)

Figure 5: Throughput evaluation on the TPC-C benchmark



(a) Throughput without distributed transactions



(b) Throughput with distributed transactions (with log scale on y-axis)

Figure 6: Throughput evaluation on the Smallbank benchmark

4. EXPERIMENTAL EVALUATION

In this section, we conduct a runtime cost analysis of our proposed adaptive logging, and compare its query processing and recovery performance against two existing logging methods. Since both traditional ARIES logging and command logging are already supported by H-Store, for consistency, we implement our distributed command logging and adaptive logging methods on top of H-Store as well. For ease of explanation, we adopt the following notations to represent the methods used in the experiments.

- **ARIES** – ARIES logging.
- **Command** – command logging proposed in [22].
- **Dis-Command** – distributed command logging method proposed in Section 2.
- **Adapt-x** – adaptive logging method proposed in Section 3. The x indicates the percentage of distributed transactions for which ARIES logs are created. There are two special cases. For x = 0%, adaptive logging is the same as distributed command logging. For x = 100%, adaptive logging adopts a simple strategy: ARIES logging for all distributed transactions and command logging for all single-node transactions.

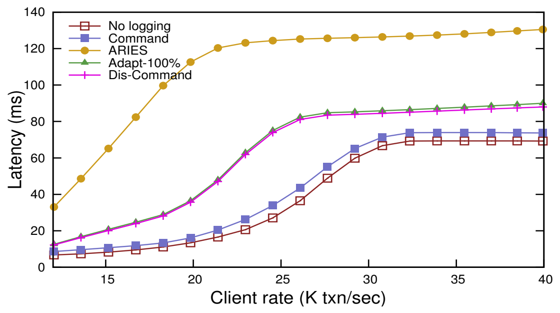
All the experiments are conducted on an in-house cluster of 17 nodes. The head node is a powerful server equipped with an Intel(R) Xeon(R) 2.2 GHz 24-core CPU and 64 GB RAM. The compute nodes are blades, each with an Intel(R) Xeon(R) 1.8 GHz 4-core CPU and 8 GB RAM. H-Store is deployed on a cluster of 16 compute nodes with the database being partitioned evenly. Each node runs a transaction site. As a default setting, only eight sites in H-Store are used in all the experiments, except in the scalability experiment (and the replication experiment in Appendix C). We use the TPC-C benchmark[2] and the Smallbank benchmark [6], with 100 clients running concurrently in the head node and each submitting its transaction requests one by one.

[2]http://www.tpc.org/tpcc/

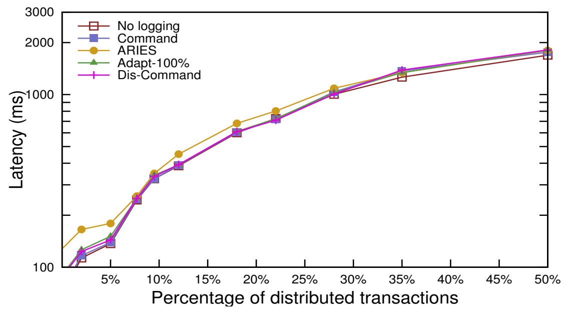
4.1 Throughput

We first compare the costs of different logging methods at runtime. For the TPC-C benchmark, we use the number of New-Order transactions processed per second as the metric to evaluate the impact of different logging methods on the system throughput. To study the behavior of different logging methods, we adopt two workloads: one contains local transactions only, and the other contains the mix of local and distributed transactions.

Figure 5a and Figure 6a show the throughput with different logging methods when only local transactions are involved. The client rate is defined as the total number of transactions submitted by all the clients per second. It varies from 10,000 transactions per second to 40,000 (resp. 100,000) transactions per second with the TPC-C (resp. SmallBank) workload in Figure 5a (resp. Figure 6a). When the client rate is low (e.g., 10,000 transaction per second), the system is not saturated and all the incoming transactions can be completed within a bounded waiting time. Although different logging methods incur different I/O costs, all the logging methods show a fairly similar performance due to the fact that I/O is not being the bottleneck. However, as the client rate increases, the system with ARIES logging saturates at around the input rate of 20,000 transactions per second for the TPC-C benchmark and around 60,000 transactions per second for the Smallbank benchmark. The other three methods, namely adaptive logging, distributed logging and command logging, reach the saturation point at around 30,000 transactions per second and 90,000 transactions per second on the TPC-C benchmark and the Smallbank benchmark respectively, which are slightly lower than the ideal case (represented as the no logging method). The throughput of distributed command logging is slightly lower than that of command logging. This is primarily due to the overhead of footprint log used in distributed command logging.



(a) Latency without distributed transactions



(b) Latency with distributed transactions (with log scale on y-axis)

Figure 7: Latency evaluation on the TPC-C benchmark

Figure 5b and Figure 6b show the throughput variation (with log scale on y-axis) when distributed transactions are involved. We saturate the system by setting the client rate to 30,000 and 90,000 transactions per second for the TPC-C benchmark and the Smallbank benchmark respectively. Meanwhile, we vary the percentage of distributed transactions from 0% to 50% so that the system performance is affected by both network communications and logging. To process distributed transactions, multiple sites have to cooperate with each other, and as a result, the coordination cost rises with the increased amount of participating sites. To guarantee the correctness at the commit phase of distributed transaction processing, all the methods adopt the two-phase commit protocol implemented in H-Store. Unlike the pure local transaction processing as shown in Figure 5a and Figure 6a, the main bottleneck of distributed processing gradually shifts from logging to network communication. Compared with a local transaction, a distributed transaction incurs additional network overhead and this reduces the effect of the logging cost.

As shown in Figure 5b, when the percentage of distributed transactions is less than 30%, the throughput of the three other logging strategies are still 1.4x higher than ARIES logging. In this experiment, the threshold x of adaptive logging is set to 100%, where the ARIES logs are created for all the distributed transactions. This is for evaluating the throughput in the worst case when adaptive logging is adopted.

As a matter of design, command logging is more suitable for local transactions with multiple updates, while ARIES logging is preferred for distributed transaction with few updates [22]. However, since the workload may change over time, neither command logging nor ARIES logging can fully satisfy all the access patterns. In contrast, our proposed adaptive logging can adapt to the dynamic change of the workload properties (e.g., distribution of the local/distributed transactions).

4.2 Latency

The average latency of different logging methods is expected to increase along the increment of client rate. Figure 7a shows that the latency of distributed command logging is slightly higher than that of command logging. However, it still performs much better than ARIES logging. Like other OLTP systems, H-Store first buffers the incoming transactions in a transaction queue, and then its transaction engine dequeues and processes them in order. H-Store adopts the single-threaded transaction processing mechanism, where each thread is responsible for a distinct partition so that no concurrency control needs to be enforced for the local transactions within the partition. When the system becomes saturated, newly arrived transactions need to wait in the queue, leading to the increment of latency.

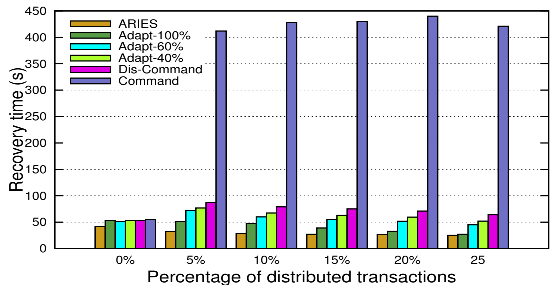
Typically, before a transaction commits at the server side, all its log entries have to be flushed to disk; after it commits, the server sends the response information back to the client. Similarly, our proposed distributed command logging materializes the command log entries and footprint log before a transaction commits. Once a transaction completes, it compresses the footprint log, which contributes to a slight delay in response. However, the penalty becomes negligible over a large number of distributed transactions as the network overhead dominates the cost. This is confirmed by the results shown in Figure 7b (with log scale on y-axis), that with an increasing number of distributed transactions, the impact on latency contributed by logging becomes secondary.

4.3 Recovery

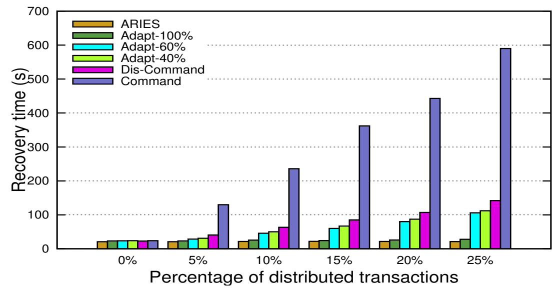
In the next set of experiments, we evaluate the recovery performance with respect to different logging methods. We simulate two scenarios of failures. In the first scenario, we run the system for one minute and then shut down an arbitrary site to trigger the recovery process. In the second scenario, each site processes at least 30,000 transactions before a random site is purposely terminated so that the recovery algorithm can be invoked. In both scenarios, we measure the time span for recovering the failed site.

The recovery times of all methods except ARIES logging are affected by two factors, namely the number of committed transactions and the percentage of distributed transactions. Figure 8 and Figure 9 show the recovery times of the four logging methods. Intuitively, the recovery time is proportional to the number of transactions that must be reprocessed upon a failure. For the first scenario of failure and its corresponding results shown in Figure 8a and Figure 9a, we observe that as the percentage of distributed transactions increases, fewer transactions are completed within a given unit of time. The reduction in the number of distributed transactions per fixed unit of time provides an offset to the more expensive recovery of distributed transactions. Consequently, as shown in Figure 8a and Figure 9a, the percentage of distributed transactions does not adversely affect the recovery time. For the second scenario of failure, we require all sites to complete at least 30,000 transactions each, and its results are shown in Figure 8b and Figure 9b. We observe that a higher recovery cost is incurred when there is an increase in the percentage of distributed transactions.

In both scenarios, ARIES logging shows the best recovery performance and is not affected by the percentage of distributed transactions, whereas command logging is always the worst performer. Our distributed command logging significantly reduces the recovery overhead of command logging, achieving a 5x improvement. The adaptive logging further improves the performance by tuning the trade-off between the recovery cost and the transaction processing cost. We provide the detailed analysis as follows.

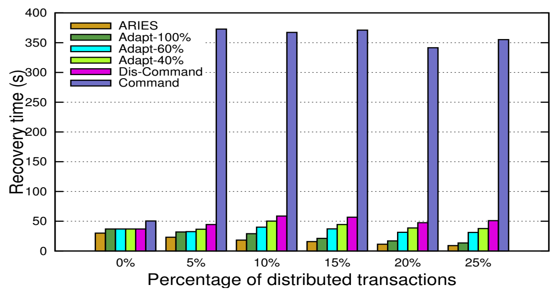


(a) 1 minute after the last checkpoint

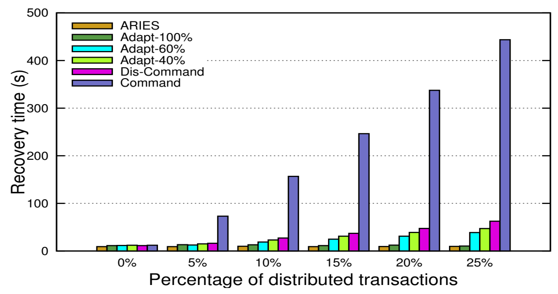


(b) After 30,000 transactions committed at each site

Figure 8: Recovery evaluation on the TPC-C benchmark



(a) 1 minute after the last checkpoint



(b) After 30,000 transactions committed at each site

Figure 9: Recovery evaluation on the Smallbank benchmark

Table 3: Number of reprocessed transactions during recovery after 30,000 transactions committed at each site

Percentage of distributed transactions	Total number of committed transactions before failure	Command	Dis-Command	Adapt-40%	Adapt-60%	Adapt-100%
0%	240031	240031	30015	30201	30087	30076
5%	239321	239321	35642	33742	32483	29290
10%	240597	240597	39285	36054	34880	30674
15%	240392	240392	42979	39687	37496	32201
20%	239853	239853	48132	43808	40912	33994
25%	240197	240197	57026	50465	46095	35617

ARIES logging supports independent parallel recovery, since each ARIES log entry contains one tuple's data image before and after each operation. Intuitively, the recovery time of ARIES logging should be less than the time interval between the checkpointing and the failure time, since read operations and transaction logics are not required to be redone during the recovery. As ARIES logging is not affected by the percentage of distributed transactions and the workload skew, its recovery time is therefore proportional to the number of committed transactions.

Command logging incurs a much higher overhead during a recovery that involves distributed transactions (even for a small proportion, e.g., 5%). This is reflected in Figure 10 , which shows the recovery time of command logging with one failed site after each site has at least 30,000 committed transactions since the last checkpoint. If no distributed transactions are involved, command logging can reach the recovery performance similar to the other methods, because transaction dependencies can be resolved within each site. Otherwise, the overhead of synchronization during recovery severely affects the recovery performance of command logging.

Distributed command logging, as illustrated in Figure 8a and Figure 9a, effectively reduces the recovery time that is required by command logging. The results in Figure 8b and Figure 9b further show that the performance of distributed command logging is less sensitive to the percentage of distributed transactions compared with command logging. However, distributed command logging incurs the cost of scanning the footprint logs for building the dependency graph. For a one-minute TPC-C workload, the time for building the dependency graph increases from 2s to 5s when the percentage of distributed transactions increases from 5% to 25%. Nevertheless, when the total recovery cost is taken into consideration, the time spent on the dependency graph construction is relatively small.

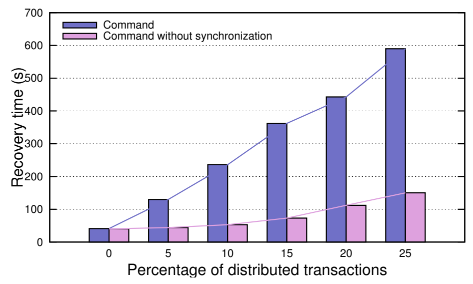


Figure 10: Synchronization cost of command logging on TPC-C benchmark

Adaptive logging selectively creates ARIES logs and command logs. To reduce the I/O overhead of adaptive logging, we set a threshold $B_{i/o}$ in our online algorithm. As a result, at most $\mathbb{N} = \frac{B_{i/o}}{W^{aries}}$ ARIES logs can be created. In this experiment, we use a dynamic threshold, by setting $\mathbb{N}$ as x percentage of the total number of distributed transactions. In Figure 8 and 9, x is set as 40%, 60% or 100% to tune the recovery cost and transaction processing cost. The results show that the recovery performance of adaptive logging is much better than command logging in all the settings. It only performs slightly worse than ARIES logging. As x increases, more ARIES logs are created by adaptive logging, resulting in the reduction of recovery time. In the extreme case, when x = 100%, ARIES log is created for every distributed transaction. As a consequence, all dependencies of distributed transactions during recovery can be resolved using ARIES logs, and each site can process its recovery independently.

Figure 11 shows the recovery performance of adaptive logging (Adapt-x) with different x values. When x = 100%, the recovery times are almost the same for all distributions, regardless of the percentage of distributed transactions. This is because all the dependencies have been resolved using ARIES log. On the contrary, adaptive logging degrades to distributed command logging when x = 0%. In this case, more distributed transactions result in higher recovery cost.

Table 3 shows the number of transactions that are reprocessed during the recovery in Figure 8b. Compared with command logging and distributed command logging (Adapt-0%), adaptive logging methods significantly reduce the number of transactions that need to be reprocessed.

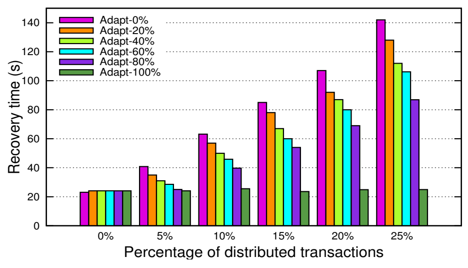


Figure 11: Recovery performance of Adapt-x on the TPC-C benchmark

4.4 Overall Performance

When commodity servers are used in a large number, failures are no longer an exception [39]. Therefore, the system must be able to recover efficiently when a failure occurs and provide a good overall performance. In this set of experiments, we measure the overall performance of different methods. In particular, we run the system for three hours and intentionally shut down a node chosen at random with a predefined failure rate. The system iteratively processes transactions and performs recovery, and creates a checkpoint every 10 minutes. Then, the total throughput of the entire system is measured as the average number of transactions processed per second over the three hours period.

Figure 12 shows the effects of failure interval on the throughput over three different distributions of distributed transactions. ARIES logging is superior to the other methods when the failure rate is very high (e.g., one failure per 5 minutes). When the failure rate is low, distributed command logging exhibits the best performance, because it is just slightly slower than command logging for transaction processing, but recovers much faster than command logging. As the failure interval drops, Adapt-100% cannot provide a comparable performance to command logging, because Adapt-100% creates an ARIES log for every distributed transaction, which is too costly in transaction processing.

4.5 Scalability

In this experiment, we evaluate the scalability of the logging methods. The percentage of distributed transactions is set to 10%, which are uniformly distributed among all the sites. Each site processes at least 30,000 transactions before we randomly terminate one site and the other sites will detect it as a failed site. It can be observed from Figure 13 that command logging does not scale. Its recovery time is positively correlated to the number of sites, because all the sites need to reprocess their committed transactions from the last checkpoint. The recovery cost of distributed command logging, which starts with a lower base compared with command logging, increases by about 50% when the number of sites is increased from 2 to 16. This is because the processing groups are larger when there are more sites involved.

The other logging methods show a scalable recovery performance. Adaptive logging selectively creates ARIES logs to break the dependency relations among the compute nodes. Therefore, the number of transactions required to be reprocessed is greatly reduced during recovery.

4.6 Cost Analysis

To study the overhead of different methods, we conduct a cost analysis using the TPC-C workload, where 5% of the generated transactions are distributed transactions. In both Figure 14 and Figure 15, the cost is presented as the percentage of CPU cycles.

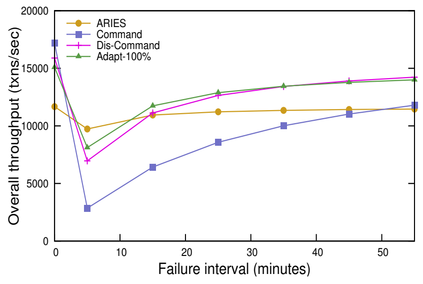
Figure 14 shows the cost at runtime. The results show that the overhead of thread blocking due to distributed transactions is the major cost in all the logging methods. For adaptive logging, online decisions can be made even when the partition is blocked by distributed transactions. So the cost of online decisions does not add much burden to the system. Similarly, footprint logging only contributes a small cost to distributed command logging and adaptive logging. The logging and footprint costs of adaptive logging slightly increase when adaptive logging uses more data logs to break dependency links. As expected, compared with the other methods, ARIES logging takes more time to construct and write the logs.

We next analyze the recovery cost of different logging methods. Figure 15 shows that command logging spends the most amount of time in performing synchronization during recovery. With adaptive logging, the composition of command logs and data logs could be controlled. The synchronization process is reduced as Adapt-x increases its x value from 0% to 100%. The clever use of data logs in breaking dependency links among distributed transactions at runtime reduces the synchronization cost during recovery. More importantly, adaptive logging is able to exploit the strengths of both ARIES and command logging to cater to different workloads.

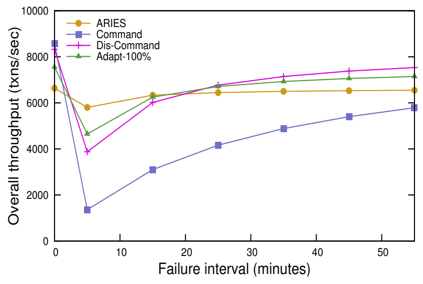
5. RELATED WORK

ARIES logging [23] is widely adopted for recovery in traditional disk-based database systems. As a fine-grained logging strategy, ARIES logging needs to construct log records for each modified tuple. Similar techniques are applied to in-memory database systems [9, 13, 14, 43].

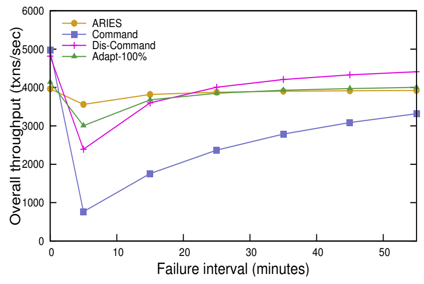
To reduce the overhead of logging, [34] proposed to reduce the log size by only recording the incoming statements. In [22], the authors argue that for in-memory systems, since the whole database is maintained in memory, the overhead of ARIES logging can be a bottleneck. Consequently, they proposed a different kind of coarse-grained logging strategy called command logging. It only records the transaction identifier and parameters instead of concrete tuple modification information.



(a) Overall throughput with 5% distributed transactions



(b) Overall throughput with 10% distributed transactions



(c) Overall throughput with 20% distributed transactions

Figure 12: Overall performance evaluation on the TPC-C benchmark

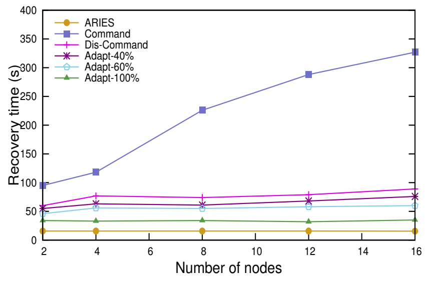


Figure 13: Scalability of logging methods on recovery for the TPC-C becnmark with 10% of distributed transactions

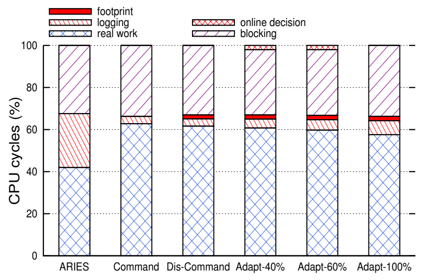


Figure 14: Runtime cost analysis on the TPC-C benchmark

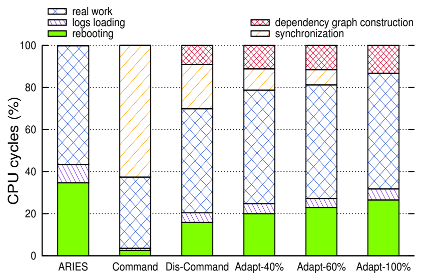


Figure 15: Cost analysis during recovery on the TPC-C benchmark

ARIES log records contain the before and after images (i.e., old and new values) of tuples. Dewitt et al [8] proposed to reduce the log size by writing only the new values to log files. However, log records without old values cannot enable the undo operation. So it needs large enough stable memory which can hold the complete log records for active transactions. They also proposed to write log records in batches to optimize the disk I/O performance. Similar techniques such as group commit [11] have also been explored in modern database systems.

Asynchronous commit strategy [2] allows transactions to complete without waiting for the log writing to finish. This strategy can reduce the overhead of log flush to some extent. However, it sacrifices the durability guarantee, since the states of the committed transactions may lose when failures happen.

Lomet et al [21] proposed a logical logging strategy. The recovery phase of ARIES logging combines physiological redo and logical undo. This extends ARIES to work in a logical setting, and the idea has been used to make ARIES logging more suitable for in-memory database system. Systems such as [16, 17] adopt this logical strategy.

If non-volatile RAM is available, database systems [4, 10, 19, 25, 29, 40] can use it to reduce the runtime overhead and speedup the recovery. With non-volatile RAM, recovery algorithms proposed by Lehman and Carey [18] could be applied.

There have been many research efforts [7, 8, 31, 33, 34, 43] devoted to efficient checkpointing for in-memory database systems. Recent works such as [7,43] focus on fast checkpointing to support efficient recovery. Usually the checkpointing techniques need to collaborate with the logging techniques and complement each other to actualize a reliable recovery process. Salem et al [35] surveyed many checkpointing techniques, which cover both inconsistent and consistent checkpointing with different logging strategies.

Johnson et al [15] identified logging-related impediments to database system scalability. The overhead of log related locking/latching contention decreases the performance of the database systems, since each transaction needs to hold the locks while waiting for its log to be materialized. Works such as [15, 27, 28] attempted to make logging more efficient by reducing the effects of locking contention.

RAMCloud [24], a key-value store for large-scale applications, replicates node's memory across nearby disks. It supports very fast recovery by carefully reconstructing the failed data from the survival nodes. However, RAMCloud does not support transactional operations that involve multiple keys. Moreover, the replication-based techniques are orthogonal to adaptive logging.

6. CONCLUSION

In the context of in-memory databases, command logging [22] shows a much better performance than ARIES logging [23] (a.k.a. write-ahead logging) for transaction processing. However, the trade-off is that command logging can significantly increase the recovery time in the case of a failure. The reason is that command logging redoes all the transactions in the log since the last checkpoint in a serial order. To address this problem, we first extend command logging to a distributed setting and enable all the nodes to perform their recovery in parallel. We identify the transactions involved in the failed node by analyzing the dependency relations and only redo the necessary transactions to reduce the recovery cost. We find that the recovery bottleneck of command logging is the synchronization process for resolving data dependency. Consequently, we design a novel adaptive logging method to achieve an optimized trade-off between the runtime performance of transaction processing and the recovery performance upon failures. Our experiments on H-Store show that adaptive logging can achieve a 10x boost for recovery while its runtime throughput is comparable to command logging.

Acknowledgments

We would like to thank Qian Lin for his suggestions that help improved the paper. We also thank the reviewers for their valuable comments. This work was in part supported by the National Research Foundation, Prime Minister's Office, Singapore under its Competitive Research Programme (CRP Award No. NRF-CRP8-2011-08). Gang Chen and Sai Wu's work was partially supported by the National Basic Research Program of China 973 (No. 2015CB-352400).

APPENDIX

A. REFERENCES

[1] MemSQL. http://www.memsql.com.

[2] Postgresql 8.3.23 documentation,chapter 28. reliability and the write-ahead log. http://www.postgresql.org/docs/8.3/static/wal-async-commit.html. Accessed: 2015-6-06.

[3] SAP HANA Wrings Performance From Wew Intel Xeons. http://www.enterprisetech.com/2014/02/19/sap-hana-wrings-performance-new-intel-xeons/.

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

[5] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *SIGMOD*, pages 761–772, 2013.

[6] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *TODS*, 34(4):20, 2009.

[7] T. Cao, M. A. V. Salles, B. Sowell, Y. Yue, A. J. Demers, J. Gehrke, and W. M. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *SIGMOD*, pages 265–276, 2011.

[8] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, pages 1–8, 1984.

[9] M. H. Eich. Main memory database recovery. In *Proceedings of 1986 ACM Fall joint computer conference*, pages 1226–1232. IEEE Computer Society Press, 1986.

[10] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *ICDE*, pages 1221–1231. IEEE, 2011.

[11] R. B. Hagmann. Reimplementing the cedar file system using logging and group commit. In *SOSP*, pages 155–162, 1987.

[12] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[13] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dalí: A high performance main memory storage manager. In *VLDB*, pages 48–59, 1994.

[14] H. V. Jagadish, A. Silberschatz, and S. Sudarshan. Recovering from main-memory lapses. In *VLDB*, pages 391–404, 1993.

[15] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: A scalable approach to logging. *PVLDB*, 3(1):681–692, 2010.

[16] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[17] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[18] T. J. Lehman and M. J. Carey. A recovery algorithm for A high-performance memory-resident database system. In *SIGMOD*, pages 104–117, 1987.

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

[20] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K.-L. Tan, and Z. Wang. Towards a non-2pc transaction management in distributed database systems. In *SIGMOD*. ACM, 2016.

[21] D. B. Lomet, K. Tzoumas, and M. J. Zwilling. Implementing performance competitive logical recovery. *PVLDB*, 4(7):430–439, 2011.

[22] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *ICDE*, pages 604–615, 2014.

[23] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 17(1):94–162, 1992.

[24] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *SOSP*, pages 29–41, 2011.

[25] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main-memory databases. CIDR, 2015.

[26] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, et al. The case for ramcloud. *Communications of the ACM*, 54(7):121–130, 2011.

[27] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[28] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: page latch-free shared-everything OLTP. *PVLDB*, 4(10):610–621, 2011.

[29] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the nvram era. *PVLDB*, 7(2):121–132, 2013.

[30] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST*, pages 17–23, 2007.

[31] C. Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica*, 1(1-4):271–287, 1986.

[32] F. Roos and S. Lindah. Distribution system component failure rates and repair times–an overview. In *NORDAC*. Citeseer, 2004.

[33] D. J. Rosenkrantz. Dynamic database dumping. In *SIGMOD*, pages 3–8, 1978.

[34] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *ICDE*, pages 452–462, 1989.

[35] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *TKDE*, 2(1):161–172, 1990.

[36] B. Schroeder, G. Gibson, et al. A large-scale study of failures in high-performance computing systems. *TDSC*, 7(4):337–350, 2010.

[37] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *VLDB*, pages 553–564. VLDB Endowment, 2005.

[38] K.-L. Tan, Q. Cai, B. C. Ooi, W.-F. Wong, C. Yao, and H. Zhang. In-memory databases: Challenges and

opportunities from software and hardware perspectives. *SIGMOD Record*, 44(2):35–40, 2015.

[39] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *SoCC*, pages 193–204. ACM, 2010.

[40] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.

[41] C. Yao, D. Agrawal, P. Chang, G. Chen, B. C. Ooi, W.-F. Wong, and M. Zhang. Exploiting single-threaded model in multi-core systems. *arXiv preprint arXiv:1503.03642*, 2015.

[42] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang. In-memory big data management and processing: A survey. *TKDE*, 27(7):1920–1948, 2015.

[43] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*, pages 465–477, Oct. 2014.

B. IMPLEMENTATION

Our distributed command logging and adaptive logging are implemented on top of H-Store [16] , a distributed in-memory database system, which supports the basic ARIES logging and command logging.

Recovery time of an in-memory database is proportional to the number of transactions recorded in the log since the last checkpoint. To reduce recovery overhead, in-memory database systems typically perform periodical checkpointing to create a snapshot in disk. Consequently, log records before the snapshot can be discarded. In our implementation, the checkpointing thread runs concurrently with the transaction processing threads, and flushes only the "dirty tuples" (i.e., tuples that have been updated since the last checkpoint) to disk. The interval of checkpointing is set to 180 seconds.

For command logging, we added one bit in its log record to indicate whether it is a distributed transaction or not. When recovering using command logging, we must synchronize the recovery process to guarantee the causal consistency. This is implemented differently in H-Store. In the single-thread architecture, a distributed transaction is intentionally blocked until all its time-dependent transactions commit, while in the single-site case, the transaction could be directly sent to the corresponding coordinator site which helps to synchronize the processing.

To build the dependency graph, our distributed command logging creates a small footprint log for each transaction, which is saved as a separate log file at each site. The footprint log is a light-weight write-ahead log which must be flushed to disk before a transaction commits. During the recovery process, one transaction may participate in multiple processing groups. However, as indicated in Algorithm 4, each transaction is recovered at most once. No redundant work is performed, and the write sets of transactions involved in multiple groups are buffered to avoid duplicated execution.

Adaptive logging makes an online decision between distributed command logging and ARIES logging. To this end, we implemented a *Decision Manager* at the transaction processing engine. The decision manager collects the statistics about transactions and maintains their dependencies. For each incoming transaction, the decision manager returns a decision to the engine which adaptively invokes different logging methods.

To further improve the performance of transaction processing in H-Store, we incorporate the following optimizations in our implementation.

1. Group commits: transactions need to flush their logs to disk before committing. Group commit batches several such flush requests together to achieve a higher I/O performance. It can also reduce the logging overhead for each transaction.
2. Dirty Column Tracking: ARIES logging records the modified tuples' old image and new image in its log records. Obviously, for tables with wide rows, it is costly to record the whole tuple. Only saving the data of those modified columns can effectively reduce the size of ARIES log. The same strategy is also adopted in our adaptive logging method.
3. Compression: it is a common technique to reduce the log size. In our implementation, the footprint log is compressed as follows. We collect the footprint log of each transaction in the form of $(txnId, tableName, tupleId)$. The information about table name and tuple ID will be duplicated in many records. So after one transaction commits, we aggregate its footprint log as $(txnId, tableName, tupleId_1, ..., tupleId_n)$. This simple operation can efficiently reduce the size of the footprint log.
4. Checkpoints: we use the checkpointing component of H-Store to do a non-blocking transaction-consistent checkpointing.

C. EFFECTS OF K-SAFETY

K-safety [37] is a replication-based technique that is often used to provide high availability and fault tolerance in the database cluster. However, it always incurs high synchronization overhead to achieve strong consistency. While this is orthogonal to the focus of the paper, it may be used to enhance logging approaches. Therefore, we conduct an experiment to study the effects of replication.

Since H-Store does not provide the K-safety feature, we measure the effects of K-safety using H-Store's commercial version VoltDB with the TPC-C benchmark. Figure 16 shows the results with different numbers of replicas. In the experiment, the cluster consists of 8 nodes and each node maintains 3 sites. We find that when K=1 (i.e., one replica is used), the throughput degrades by 46% and latency increases by 63% compared to the one without replicas. When K=7 (i.e., system could provide availability even there is only one healthy node remaining), the throughput degrades by 82.6% and the latency increases by 483%. In summary, system's performance degrades significantly when replication is enabled.

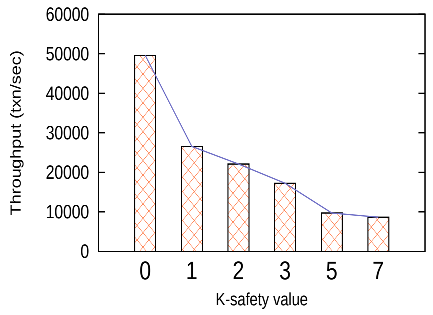
In terms of availability, K-safety shows good performance. However, it still depends on logs to recover when more than K nodes fail simultaneously. We conducted experiments to measure the availability in Appendix E.2 and Appendix E.3.

D. RUNTIME ANALYSIS

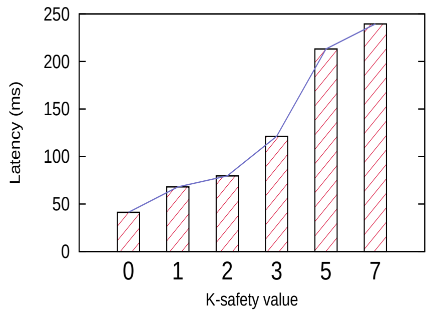
In this set of experiments, we conduct a cost analysis of the online algorithm and the in-memory index in our proposed adaptive logging.

D.1 Online Algorithm Cost of Adaptive Logging

In Figure 17, we analyze the computation cost of every minute by showing the percentage of time taken for making online decisions and processing transactions. Overheads of the online algorithm increase when the system runs for a longer time, because more transaction information is being maintained in the in-memory index. However, we observe that it takes only 5 seconds to execute the online algorithm in the 8th minute, and the main bulk of the time is still spent on transaction processing. In practice, the online decision cost would not grow in an unlimited manner as it is bounded by the checkpointing interval. Since the online decision is



(a) Throughput evaluation



(b) Latency evaluation

Figure 16: Effects of K-safety on the TPC-C benchmark

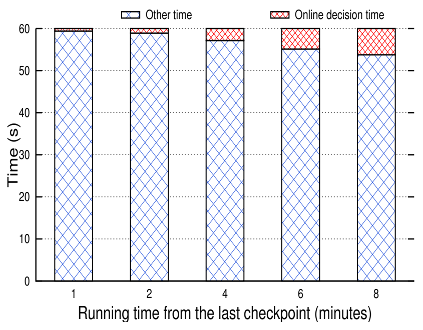


Figure 17: Cost of online decision algorithm on the TPC-C benchmark

made before the execution of a transaction, we could multitask the computation of online decisions while the transaction is waiting in the transaction queue to further reduce the latency.

D.2 Online Algorithm vs Offline Algorithm

The offline algorithm assumes that the distribution of transactions is known upfront, and such information can be used to generate an optimal logging strategy for performance comparison purposes. The offline algorithm can be considered as the ideal case, but it is not realistic in real implementation. For the online algorithm, we exploit historical statistics to predict the distribution of future transactions so that the system can dynamically make a decision on a logging plan.

We compare the performance of the online and offline algorithms in Figure 18. In this experiment, we disabled the checkpoints. For the offline algorithm, we ran the same workload twice. For the first run, we recorded the transaction distribution information, which is used as priori knowledge in the second run. As shown in Figure 18, the throughput of the online algorithm slightly decreases when the system runs for a longer time, since more information is maintained in the in-memory index, causing the overhead of logging selection to increase. It degrades about 6.24% at the 8th minute after about 3.8 million transactions are processed. In practice, the online decision cost will not keep growing as it is bounded by the checkpointing interval.

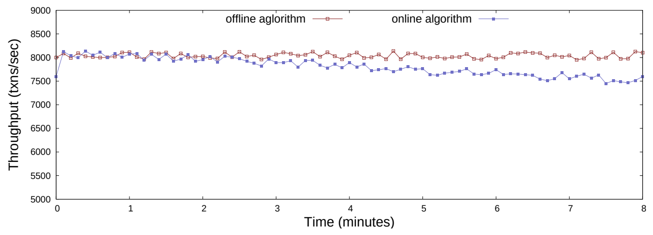


Figure 18: Comparison between online and offline algorithms on the TPC-C benchmark

D.3 Accuracy of the Online Algorithm

In Figure 19, we evaluate the accuracy of online algorithm by comparing with the ideal offline algorithm and a random algorithm. The result of the offline algorithm which assumes that it knows all the transactions is used as the ground truth in the comparison. For the random algorithm, if system still has I/O budget, it chooses the ARIES log or command log at random; otherwise, it sticks to command logging. The approximate estimation of our online algorithm is able to produce logging plans that are similar to those produced by the offline algorithm. If the I/O budget can only support building ARIES logs for 20% distributed transactions, the approximate online algorithm achieves 80% prediction accuracy.

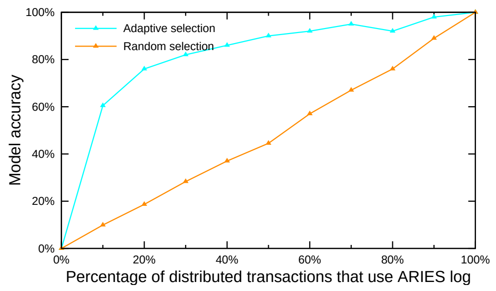


Figure 19: Accuracy of different selection models on the TPC-C benchmark

D.4 Effects of Transaction Pattern

We next conduct experiments on the TPC-C benchmark to evaluate the effects of transaction pattern on our logging selection algorithm. After the system becomes saturated, we change the hot item set at the 10th minute. In Figure 20, we measure the accuracy of our online algorithm as well as the random algorithm. As expected, the accuracy of the proposed online algorithm drops significantly when the transaction pattern changes dramatically, since it uses a wrong (old) statistical distribution. In our implementation, adaptive logging updates the statistical distribution every minute. As can be observed, the online algorithm continues to provide a good prediction as soon as it is able to use the new statistical distribution (after the update).

D.5 Effects of the In-memory Index

The main purpose of the in-memory index is for estimating the cost/benefit of using either the ARIES log or the command log. Asynchronous update of the in-memory index is used to reduce

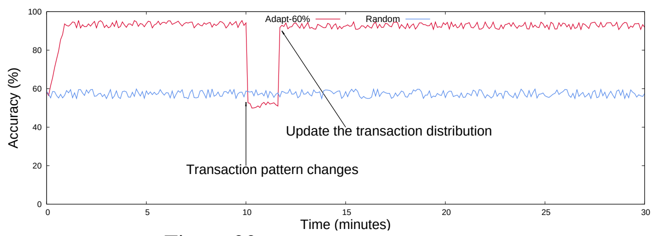
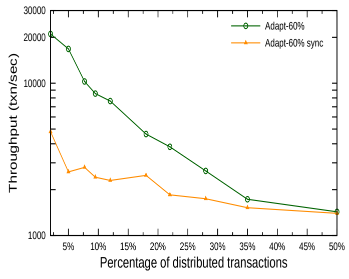


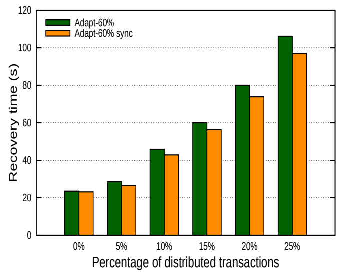
Figure 20: Effects of transaction pattern

the maintenance overhead. Indexes of a batch of transactions are cached locally and periodically synchronized with the master node. Since the in-memory index is used mainly for estimation, it is unnecessary to require the index in the master node to be strictly consistent (eventual consistency is sufficient).

We have also conducted experiments on the TPC-C benchmark to show the benefits of asynchronous update by comparing the two strategies: asynchronous update and synchronous update. Synchronous update requires the transaction to update the in-memory index in the master node before it commits. For asynchronous update, transaction only updates the cached indexes in the processing nodes. The processing nodes will synchronize the indexes in the master node periodically. As shown in Figure 21, synchronous update is very costly and dramatically affects the performance of transaction processing. Further, synchronous update only leads to a marginal improvement in the recovery process. This observation confirms that the asynchronous approach is a more efficient approach.



(a) Throughput at runtime



(b) Recovery time when 30,000 transactions commit at each site

Figure 21: Effects of asynchronous update on in-memory index

E. RECOVERY ANALYSIS

We conduct a set of experiments to evaluate the system performance in different failure scenarios.

E.1 Master-Node Failure

The master node maintains the in-memory index which is used to help estimate the cost/benefit of using the ARIES log and command log. If the master fails and the in-memory index is lost, adaptive logging always sticks to command logging, because our adaptive logging model enforces a constraint such that ARIES log will not be preferred unless the benefit of building the ARIES log is larger than a predefined threshold. If the in-memory index is absent, it is hard to retrieve transaction's time-dependent transactions, all ARIES logs are estimated to be not beneficial enough. To address the issue, our system caches the in-memory index in each processing node. It can be updated in an asynchronous manner. In the event when the master node fails, the system can still switch between the two logging approaches.

As discussed in Appendix D.5, the in-memory index is updated in an asynchronous manner. As shown in Figure 22, when the master node fails, the transaction coordinator can still apply the adaptive approach to select a logging strategy using its local copy of in-memory index. The throughput slightly increases because there is no communication between the master and slave nodes. After a new master node is elected, all nodes will synchronize their local indexes with the new master node, which incurs high communication cost and leads to the throughput drop. The index synchronization only lasts for a few seconds. After the index in the master node is synchronized and the system is fully recovered, the throughout increases until it becomes steady.

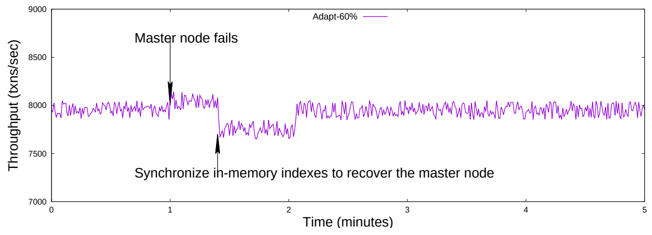


Figure 22: Throughput evaluation on TPC-C benchmark when the master node fails

E.2 Single-Node Failure

In Figure 23, we simulate a single-node failure by randomly selecting a node and killing its running processes. Systems with ARIES logging and command logging have to halt (the throughput drops to zero) until the failed node is fully recovered. On the contrary, system with K-safety can continue the transaction processing. Its throughput even slightly increases, since the number of replicas that needs to be synchronized decreases. System with adaptive logging can process new transactions and do the recovery simultaneously. As discussed in Section 2.4, the system first constructs the dependency graphs based on the lightweight footprint log. With the dependency graphs, the database in the failed node can be recovered on demand while the surviving nodes continue with the processing. The throughput gradually increases as the database in the failed node is being recovered.

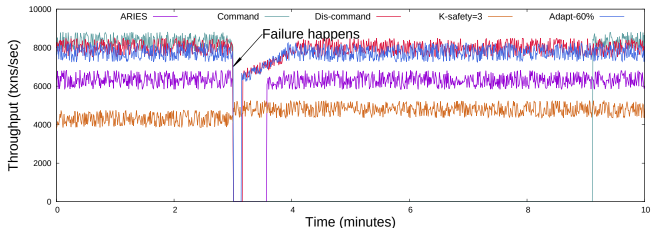


Figure 23: Throughput evaluation on the TPC-C benchmark when a single node fails

E.3 Entire-Cluster Failure

Figure 24 shows the results when the entire cluster fails. Unlike the case of a single-node failure, system with K-safety cannot retain system availability and durability if the entire cluster fails. Thus, it still has to rely on logs to perform the recovery (e.g., command logging is enabled for K-safety in this experiment).

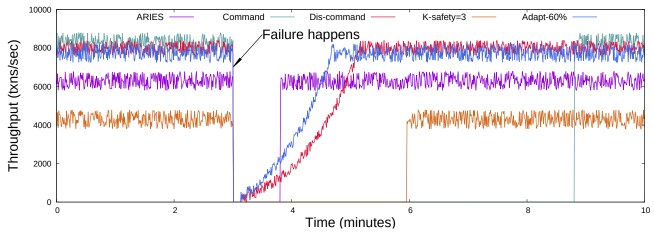


Figure 24: Throughput evaluation the TPC-C benchmark when the entire cluster fails