On-Demand Recovery in Middleware Storage Systems

Lásaro Camargos, *Unicamp, Brazil*
Fernando Pedone, *USI, Switzerland*
Alex Pilchin, *EPFL, Switzerland*
Marcin Wieloch, *USI, Switzerland*

***Abstract*—This paper presents a recovery architecture for in-memory data management systems. Recovery in such systems boils down to solving two problems: retrieving and installing the last committed image of the crashed database on a new server and replaying the updates missing from the image. We improve recovery time with a novel technique called *On-Demand Recovery*, which removes the need to replay all missing updates before new transactions can be accepted. We have implemented and thoroughly evaluated the technique. We show in the paper that in some cases On-Demand Recovery can reduce recovery time by more than 50%.**

***Keywords*-recovery; database; optimistic techniques**

I. Introduction

In the past years middleware-based data management systems have become very popular. At the back-end of most such systems lies a database engine. To meet the ever growing demand for performance and scalability in these environments, some researchers in the database community have come out to voice criticism to the one-size-fits-all model of modern-day databases and argued for a redesign, which would take into account application-specific characteristics and hardware trends [1], [2].

On the one hand, application programs in multi-tier architectures create transactions typically by instantiating parameterized templates, defined offline, as opposed to submitting ad hoc statements defined on-the-fly. As a consequence, some workload characteristics can be inferred before the execution (e.g., transaction access patterns). On the other hand, plummeting hardware prices have made powerful clustered environments largely affordable. In what concerns data management systems, it is believed that in some years all but the largest databases will fit in the aggregated memory of the servers of medium-size clusters [3]. Accounting for application and hardware specifics requires revisiting key database concepts, such as recovery.

This paper presents a recovery architecture for clustered in-memory databases. Recovery in such systems boils down to solving two problems: (i) retrieving and installing the last committed image of the crashed database on an available server and (ii) replaying the updates missing from the image. Different recovery protocols may solve these problems in different ways, trading their inherent overheads. In Sprint, for example, a middleware-based in-memory data management system [4], database images and logs are created and stored by remote servers—in fact, in-memory database servers do not even have to be attached to a disk unit.

Recovering a crashed database server requires retrieving and installing its last committed image on a new server, and replaying the log of missing updates. Despite the larger size of the database image, with respect to the log of missing updates, a substantial fraction of recovery time in Sprint involves bringing the recovered image up-to-date by processing missing updates—in part this is due to the fact that replaying missing updates requires re-executing statements. An obvious solution to this problem is to create remote images more often. Creating an image, however, interferes with the normal system execution and therefore should be done sparingly.

To reduce recovery time, this paper introduces an *On-Demand Recovery* protocol, which exploits application information (i.e., pre-defined transaction templates) to avoid replaying all missing updates before new transactions can be accepted. The idea is conceptually simple: a new transaction can be executed as soon as all the missing updates it depends on have been replayed. On-Demand Recovery identifies such dependencies and selectively replays the missing updates only. As a result, the system can receive and execute new transactions earlier, increasing its availability.

Performance experiments reported in the paper show that On-Demand Recovery can reduce recovery time by more than two times. In one scenario, for example, we show that it takes 12 seconds to recover a database of 2.5 GBytes running TPC-C transactions. In this context, even if one conservatively assumes that a server is rebooted a dozen times a year, the resulting availability will still be larger than "five nines" (i.e., five minutes of downtime per year). Although we illustrate On-Demand Recovery in Sprint, the technique is general enough and could be used in different contexts. For example, On-Demand Recovery could be also used to speed up the recovery of a traditional standalone database server, although we do not develop this point further in the paper.

The remainder of the paper is structured as follows. Section II describes the system architecture. Section III introduces On-Demand Recovery and Section IV discusses how it was implemented. Section V contains the performance evaluation. Section VI reviews related work, and Section VII concludes the paper. The appendix discusses the correctness of On-Demand Recovery.

II. System architecture

The recovery system we propose is based on the Sprint architecture [4] (see Figure 1). In this section we present the main assumptions and components of this architecture.

A. Background

The system is composed of a cluster of servers that communicate by message passing only (i.e., there is no shared memory). Communication assumes FIFO channels: messages are received in the order they are sent.

We assume that servers are *fail-stop*: each server halts in response to a failure and a server's halted state can be detected by operational servers. We do not consider malicious failures (i.e., Byzantine failures), in which servers may present arbitrary behavior.

There are two types of servers: *physical servers*, part of the hardware infrastructure, and *logical servers*, the software component of the system. Logical servers can be of four types: Edge Servers, Data Servers, Durability Servers, and Recovery Servers.

The database is partitioned among the Data Servers. If desired (e.g., for performance), data can be replicated in multiple Data Servers. On-Demand Recovery, however, does not assume that data is replicated in multiple Data Servers.

B. Edge Servers

Edge Servers receive query requests from the clients, break them into partial queries, taking the database partitioning into account, and forward the partial queries to the appropriate Data Servers. Data Servers process the partial queries locally and reply to Edge Servers. Edge Servers then post-process the partial results to answer the original queries and reply to the clients. Partitioning information is maintained as soft state by each Edge Server; all permanent state is stored at the Durability Servers. As a consequence, creating a new Edge Server or recovering a crashed one is straightforward.

C. Data Servers

Data Servers execute transactions, i.e., sequences of SQL queries terminating with a commit or an abort statement. Each transaction has a unique identifier. We assume the traditional ACID properties of transactions: atomicity, consistency, isolation (i.e., serializability), and durability [5].

Data Servers keep partitions of the data set in in-memory databases for fast transaction processing. Although Data Servers are oblivious to the global data partitioning, they are aware of which Data Servers are involved in the transactions they execute. To terminate a transaction consistently and atomically, Edge Servers require all Data Servers involved in the transaction—those which executed the transaction's partial queries—to run a Non-Blocking Atomic Commitment (NBAC) protocol.

In the Atomic Commitment problem, all participants must vote and agree on committing or aborting a transaction; the transaction is committed iff all participants vote to commit it. NBAC solves a relaxed version of the problem in which the decision may be abort if some participant is suspected to crash, despite all participants voting to commit. The protocol is non-blocking because progress is guaranteed despite the failure of any participant (i.e., Data Server) or the coordinator (i.e., relevant Edge Server). If the coordinator or a Data Server fails before committing the transaction, it will be aborted by the remaining Data Servers.

D. Durability Servers

Durability Servers play a key role in the execution of NBAC. Once requested by the Edge Server, Data Servers send their votes to the Durability Servers to be ordered and logged. Durability Servers implement the Paxos protocol [6] to totally order all votes received. The ordered votes for a committing transaction are seen by all Data Servers involved in the transaction, which locally abide to the outcome of each transaction they have taken part in. Alongside the votes for a given transaction, Data Servers also forward the updates (i.e., SQL statements) that they have performed in the context of the committing transaction to the Durability Servers. Then, the Durability Servers, by means of the Paxos protocol, ensure that the votes and corresponding updates are stored in stable storage.

In case of a Data Server crash, a new Data Server is created to replace it. In principle, the new Data Server can be brought up-to-date by replaying all the updates that the crashed server executed, in commit order. Missing updates are retrieved from the Durability Servers in the right order by scanning the Paxos logs. In practice, Durability Servers perform several optimizations in order to speed up recovery, as described later.

E. Recovery Servers

Recovery Servers speed up recovery without compromising normal transaction processing. Their role is twofold: (a) minimizing the extra work that Durability Servers have to perform during normal processing and recovery; and (b) increasing system availability by speeding up the creation of a new Data Server to replace a failed one. We explain how these goals are achieved in the next section.

III. Recovery architecture

A. Baseline approach

In the baseline approach, to recover a Data Server the new server first fetches and installs the latest image created of the crashed server and then applies the tail of the log corresponding to the committed transactions not included in the image.

To reduce the work done by the Durability Servers during normal execution and the amount of information to

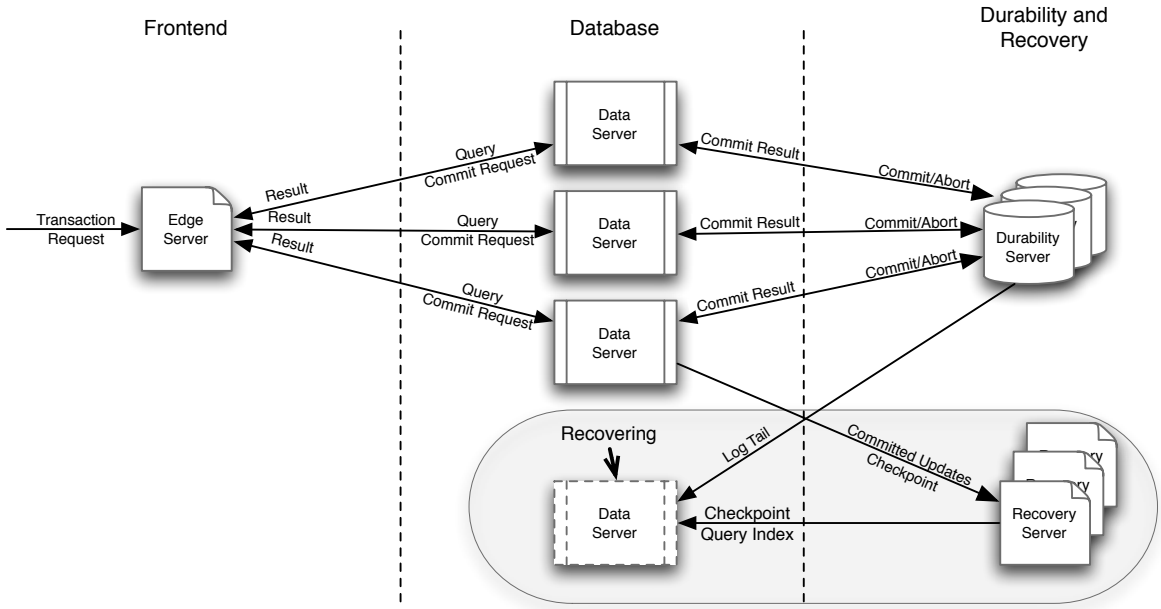


Figure 1. Transaction processing architecture

be retrieved from them upon recovery, Data Servers take remote checkpoints of their in-memory databases. These checkpoints are stored on the Recovery Servers and taken asynchronously to minimize performance loss at the Data Servers.

Since Data Servers rely on a local database engine for transaction execution, we assume that the engine has a mechanism to take checkpoints asynchronously (e.g., fuzzy checkpoints [7]) and provides some control over where these checkpoints are stored. In particular, they should be kept remotely, and not on the server's local disk. We discuss in Section IV how this mechanism was implemented in middleware, with little database support.

In addition to checkpointing their state, Data Servers also forward to Recovery Servers mutative statements (e.g., update, insert, delete) of committed transactions, to which hereafter we refer as *committed updates*. To recover from a failure, a small *log tail* may have to be recovered from the Durability Servers as well. This log tail contains transactions that have been committed, but whose updates have not reached the Recovery Servers because of the crash of the Data Server. The log tail is expected to be small since Data Servers forward committed updates to Recovery Servers upon transaction commit.

During recovery, the new server installs the last checkpoint of the crashed Data Server and asks the Recovery Server for the list of committed updates—notice that since Data Servers use FIFO channels to send updates to Recovery Servers, the list may be incomplete, but it does not contain holes. After receiving the list of committed updates from a Recovery Server, the new server sends a request to a Durability Server to fetch the log tail, which will be used to complete the list of committed updates.

Recovery is complete and new transactions can be processed once the checkpoint is installed and all missing committed updates have been executed.

B. On-Demand Recovery

On-Demand Recovery reduces recovery time by accepting new transactions before processing all updates in the list of committed updates. To implement this, we replace the list of committed updates by a structure called *Query Index*. The Query Index is a directed acyclic graph in which vertices represent committed updates and edges represent their *dependencies*. Statements that change the same data items are ordered in the index according to the commit order of the transactions to which they belong. The index is kept in memory for performance and does not grow indefinitely: on a successful checkpoint, updates included in the checkpoint are removed from the Query Index.

The Query Index is used to look up update operations that must be executed before new incoming queries—precisely, because they conflict with these queries. The motivation for this scheme is to amortize the cost of recovery across incoming queries. Moreover, to minimize the work to be done for future queries and ensure that all updates in the Query Index will be executed against the database, a background task continuously applies queries from the index

to the database. Thus, eventually the index will be emptied, speeding up the execution of new queries.

The design of our indexing structure was guided by the observation that transactions in modern multi-tier architectures are built from *parameterized templates* (e.g., SQL statements), defined before the system starts to accept transactions and instantiated during the execution. As a result, the query space is well-defined and dependencies can be determined offline, either automatically, by parsing transaction templates, or manually, if the set of templates is reasonably small. In the case of the TPC-C benchmark, used in the evaluation presented in this paper, we initially extracted the needed information from the transaction templates manually, and then confirmed the results using an SQL Inspector tool, developed in our group [8].

The Query Index builds on two concepts:

- The **Query Descriptor** of query Q indicates which attributes in Q should be used for storing it in the Query Index and retrieving its dependencies from the index.
- The **Cover Graph** of query Q is the set of queries in the index upon which Q depends together with their dependencies (i.e., it is a subgraph of the Query Index).

C. Query Index by example

We illustrate the specifics of the indexing technique with a simple example. Assume a table Account with two fields, account number (aid), used as the primary key, and balance (bal). We consider the query types below, upon which the following transaction templates are defined.

Query types:

Query	SQL statement
$Rate(r)$	update Account set bal=bal*r
$Ins(x, v)$	insert into Account(aid,bal) values (x, v)
$Add(x, v)$	update Account set bal=bal+v where aid=x

Transaction templates:

Transaction	Defined as	Description
AddInterest(r)	$Rate(r)$	increase balance of all accounts
NewAccount(x, v)	$Ins(x, v)$	create new account
Transfer(x, y, v)	$Add(x, -v)$, $Add(y, v)$	transfer v from account x to y

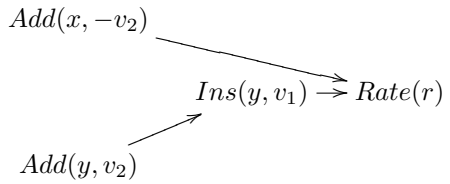
From the query types above, the descriptors below will be built.[1] Query $Rate$ depends on any other queries of types $Rate$, Ins, and Add, despite their account numbers, if any. Queries Ins and Add will be indexed based on their account number. Ins depends only on $Rate$; Add depends on $Rate$ and on any other query of type Ins and Add defined on the same account number. In more complex workloads (e.g., TPC-C), we can also incorporate information about tables, columns and values accessed by the queries to establish dependencies (c.f. Section IV).

[1]Notice that our notion of dependency is quite conservative: one query depends on another if they require update locks on common rows—we assume that inserts lock the complete table. While more sophisticated notions of dependency could be used, we observed experimentally that even a simple one can provide good results.

Query descriptors:

Query	Index	Depends on
$Rate(r)$	–	$Rate(\star)$, $Ins(\star, \star)$, $Add(\star, \star)$
$Ins(x, v)$	x	$Rate(\star)$
$Add(x, v)$	x	$Rate(\star)$, $Ins(x, \star)$, $Add(x, \star)$

Assume now that the Query Index contains four queries: $Rate(r)$, $Ins(y, v_1)$, $Add(x, -v_2)$, and $Add(y, v_2)$, created after AddInterest(r), NewAccount(y, v_1), and Transfer(x, y, v_2) were executed, in this order. In the Query Index, these queries are stored as depicted next.



A new transaction Transfer(z, x, v_3) will create queries $Add(z, -v_3)$ and $Add(x, v_3)$, with cover graphs $Rate(r)$ and $Add(x, -v_2) \rightarrow Rate(r)$, respectively. Thus, before $Add(z, -v_3)$ can be submitted, $Rate(r)$ must be executed against the database; $Add(x, v_3)$ can be only executed after $Rate(r)$ and $Add(x, -v_2)$ have been executed, in this order.

D. Synchronizing recovery sources

To restore the state of a crashed Data Server, a new server uses information from: (a) the last checkpoint taken by the crashed server, (b) the Query Index, and (c) the log tail. Recovery must ensure that the combination of these data sources will not lead to lost and duplicated transactions.

We use a lazy mechanism to remove transactions from the Query Index after a checkpoint. While this mechanism avoids lost transactions, it may lead to duplicates (i.e., transactions that are both in a checkpoint and in the Query Index). Removing duplicates from the Query Index is based on two characteristics of the system:

- Data Servers have a fixed pool of *working tasks* (i.e., those that submit transactions to the local in-memory database), whose number is defined by the Data Server *concurrency level*, a parameter of the system.
- Data Server checkpoints are *task-prefix consistent*: for every working task, if a transaction committed by the task is included in the checkpoint, then all transactions previously committed by the task are also included.

Tracking duplicated transactions is a two-step procedure: First, we augment the in-memory database with a vector with one entry per working task, implemented as an extra database table. Each entry in this vector stores the identifier of the last transaction committed by the task, and is updated

by an additional operation automatically added to each mutative transaction. Second, each update in the Query Index contains the identifier of the task that executed it. Since there is one entry per task in the vector and each task modifies its own entry only, this mechanism does not introduce data contention—although it introduces an additional small table and one more operation per transaction.

To remove duplicates, after installing the last checkpoint, the new Data Server retrieves the vector of transaction identifiers and removes all updates from the Query Index whose identifiers are in the vector, together with all the updates that precede them in the index.

A similar procedure is used to avoid duplicates in the log tail provided by the Durability Servers: The Query Index keeps a vector of transaction identifiers, one per Data Server working task, containing the identifier of the last committed transaction whose updates have been stored in the index. When the Data Server requests the log tail from a Durability Server, it includes in the request its vector of transaction identifiers.

To build the log tail for a Data Server, the Durability Server scans its log twice: once backward looking for transactions in the transaction identifiers vector, and once forward looking for transactions that succeed those transactions in the log; only these transactions enter the log tail.

IV. Implementation

We integrated our protocols and the Recovery Servers into Sprint [4], a middleware infrastructure implemented in Java. Data Servers run MySQL with InnoDB storage engine. InnoDB is a disk-based storage engine. We ensure that data processing is "in memory" by carefully selecting data sets that fit the main memory of each Data Server and disabling synchronous disk writes.

In order to perform remote checkpoints, we configured Data Servers to use Recovery Server disks using *ATA over Ethernet* (AoE) [9]. Each Data Server has at the Recovery Server a separate disk partition and a dedicated AoE server daemon (*vbladed*) making the partition accessible over the network. A Data Server mounts a remote partition into its file system and uses it as InnoDB data and log directories.

InnoDB does not provide direct control over checkpoints. In our experiments we set the maximum log file size to bound checkpoint intervals and periodically discard the contents of the Query Index. Checkpoints can be observed by monitoring the Log Sequence Number of the last checkpoint with MySQL's *show engine InnoDB status* command. Moreover, upon restart, we must instruct MySQL to install the last checkpoint without applying its log tail to it, otherwise most of the updates in the Query Index would be conservatively executed against the database. InnoDB does not provide this functionality. Thus, in the experiments we used a trimmed log containing only the last checkpoint.

We divided the TPC-C database into logical data items, each one indexed by a key composed of table, column, and row identifiers. For vertical partitioning, we divided tables *warehouse*, *customer*, and *stock* into two parts: one with read-only columns and the other with the rest. Tables *new_order*, *order*, and *order_line* do not make this distinction, while table *district* defines two columns that can be updated as separate partitions. For horizontal partitioning, the key is mostly based on values of *warehouse_id* and *district_id*, however it is extended with additional attributes: for the *stock* table the extra attribute is *item_id*; for tables *order*, *order_line*, and *new_order* it is *order_id*. We considered neither table *history* nor table *item*; the former is only modified with inserts, and the latter is a read-only table.

The Query Index uses three data structures (see Figure 2): a linked list of committed updates, the dependency graph, and a hash table of data items. A transaction consists of its identifier, the identifier of its working task, and a sequence of updates. The Query Index stores transaction updates in the linked list, ordered according to the execution order. Vertices in the dependency graph are pointers to the actual updates; edges describe dependencies between them. The hash table is the starting point for the dependency check: it maps a data item identifier to the most recent update in the dependency graph affecting this data item.

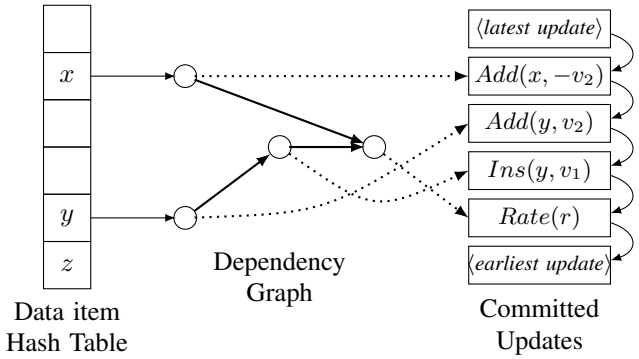


Figure 2. Query Index data structure

V. Evaluation

A. Experimental setup

We ran all the experiments in a cluster of Dell SC1435 servers equipped with two dual-core AMD-Opteron 2.0 GHz processors, 4 GB of main memory, and a 73 GB 15krpm SAS disk. The servers are interconnected through an HP ProCurve2900-48G Gigabit Ethernet switch. We used up to 18 nodes in the experiments. Each logical server ran on a dedicated node, and Edge Servers were integrated with the workload generator, which ran multiple client threads. There were 3 nodes hosting Durability Servers, and these had the on-disk write cache disabled to perform synchronous disk writes. The remaining nodes were assigned to the Recovery

Server (1 node), Data Servers (1 to 6 nodes), Edge Servers (1 to 7 nodes), and one additional node collected data from the workload generators.

B. TPC-C benchmark analysis

We initially analyzed query dependencies in TPC-C. For this purpose we created a Query Index with approximately 69000 updates, corresponding to the execution of 4800 TPC-C transactions, and checked it against a recovery trace with approximately 6000 new incoming TPC-C transactions. For each transaction in the trace we counted its dependencies in the Query Index before removing the dependencies from the index, as it would happen in an execution of the system.

Figure 3 shows the ordered dependency ratio of transactions in the trace. The dependency ratio of a transaction is calculated as the number of dependencies the transaction has in the Query Index divided by the total number of updates currently in the index. The results show that less than 0.6% of the transactions in the trace (35 transactions) have dependencies within the range of 1%–5.2%; approximately 4% are within the range of 0.1%–5.2%; and more than 95% of new incoming transactions conflict with less than 0.1% of the updates in the Query Index.

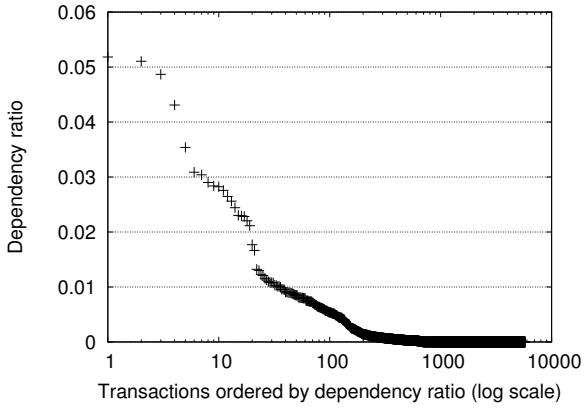


Figure 3. Dependency ratio in TPC-C

In Figure 4 we considered how the Query Index evolved during the execution. The first 1000 transactions in the trace bring the Query Index to less than half of its original size. We also observed that at the end of the execution, about 11000 updates remained in the index, that is, none of the transactions in the trace depended on these updates. It turns out that those are insert statements accessing TPC-C's *history* table, never read nor modified by other transactions, however required by the benchmark. In real runs of the system (i.e., in the rest of our experiments), such statements would be submitted to the database by the background task.

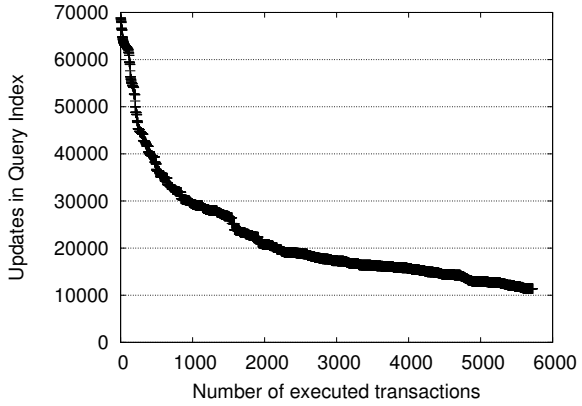


Figure 4. TPC-C updates in the Query Index

C. Recovering a Data Server

We compared On-Demand recovery with the baseline using four scenarios (see Table I). In each scenario we varied the frequency of checkpoints and the size of the Query Index. In scenario (i), for example, the index contained approximately 69000 updates from 4800 transactions, corresponding to 17.6 MBytes of data. In this configuration we set the log size to 5 MBytes, resulting in a throughput of approximately 73 transactions per second. Notice that although more sparse checkpoints lead to better performance, they also result in larger indexes, since garbage collection of dependencies in the Query Index is less frequent.

Scenario	(i)	(ii)	(iii)	(iv)
Database size	2.5 GBytes			
Query Index				
number of transactions	4.8k	10k	30k	54k
number of updates	69k	141.6k	424.1k	762k
total size (MBytes)	17.6	36.1	108.1	194.2
Throughput (TPS)	73	81	80	84
Log file size (MBytes)	5	10	25	50
Checkpoint frequency (min)	≈ 1	≈ 2	≈ 6	≈ 10

Table I
RECOVERY SCENARIOS

Table II presents the approximate recovery times in seconds of each technique. The recovery time is the time it takes for a recovering Data Server to start accepting new transactions. For baseline we show the breakdown of its two main activities. Installing a remote checkpoint is a relatively quick operation: all what it takes is to instruct MySQL to bootstrap using the remote disk located at a Recovery Server. Consequently, retrieving the log tail amounts to most of the time reported for this activity, which is mainly related to serializing and de-serializing the log structure. On-Demand Recovery addresses the main source of overhead

in recovering a Data Server. In the most favorable setup, scenario (i), it leads to a recovery speedup of 2.1 times. Although the technique is less effective when checkpoints are sparse, due to the growth of the Query Index, it was beneficial in all the scenarios we considered.

Scenario	(i)	(ii)	(iii)	(iv)
Baseline	**25s**	**34s**	**109s**	**237s**
install checkpoint and retrieve committed updates	3s	6s	20s	37s
apply committed updates	22s	28s	89s	200s
On-Demand Recovery	**12s**	**24s**	**88s**	**209s**
Improvement	2.1×	1.4×	1.2×	1.1×

Table II
RECOVERY TIMES

D. Recovery history

We take now a closer look at a recovering Data Server. Figures 5–8 show the throughput and response time during recovery in scenarios (i) and (iii). The graphs also show the number of dependencies (i.e., committed updates) applied to the database during recovery. Notice that the graphs have a primary and a secondary y axis; throughput and response time are reported in the primary axis (left side of the graph); dependencies in the secondary axis (right side of the graph). From the graphs, performance is driven by the fact that pages should be loaded into main memory (until we reach the "in-memory effect", when all pages are loaded) and dependencies should be applied to the database. In scenarios (i) and (iii), all dependencies are applied after 42 and 164 seconds, respectively.

The drops in throughput and corresponding spikes in response time are related to the checkpoint mechanism used by InnoDB. InnoDB implements a fuzzy checkpoint technique, which means that in principle performance should not suffer during checkpoints. In reality, however, InnoDB flushes dirty pages to disk based on ranges of their Log Sequence Numbers (LSNs), assuming a somehow uniform distribution of LSNs. Under some workloads, such as TPC-C, a large number of dirty pages fall on the same range, leading InnoDB to flush most of its buffer pool pages. In addition to the fuzzy checkpoint mechanism, InnoDB uses two log files on a rotating basis. Before a file can be reused, InnoDB has to ensure that the database image on disk contains all pages logged in the file, forcing modified pages to be flushed.

E. Recovery Servers

Figure 9 considers the number of Data Servers that can be handled by a single Recovery Server in scenario (i) for moderate (i.e., 50 TPS) and high load (i.e., 73 TPS). In these experiments we created six disk partitions in one Recovery Server and run up to 6 AoE servers, each one serving one Data Server. Under moderate load a single Recovery Server can serve up to 3 Data Servers with a loss of less than 9% in throughput; under high load 2 Data Servers can be served with a throughput loss of 7% at most.

VI. RELATED WORK

In this paper, we have introduced a remote recovery architecture, which allows the recovery of a crashed Data Server without relying on its local state. The recovered state comes from Recovery Servers (i.e., database checkpoint and missing updates) and Durability Servers (i.e., log tail). Even though in principle it is possible to rebuild the state of a Data Server from the logs stored at the Durability Servers only, we can speed up the procedure by using remote checkpoints and applying a small log tail. Similar ideas have been taken by other systems, such as Big Table [10], where major "contractions" (i.e., checkpoints) are durably stored in GFS [11]. A major contraction (i.e., applying log tail) is also performed on recovery, prior to restart. The difference between our approach and these works is in how we store and structure the log tail (i.e., in the memory of the Query Index for faster recovery), in how we do recovery (i.e., on demand), and in the richer query semantics that we can support (e.g., SQL-based databases).

Several works have used group communication to implement full database replication, and some of these explicitly discuss recovery (e.g., [12], [13], [14]). In [12] the authors discuss how to bring a crashed site (or a brand new one) back up without stopping transaction processing. The recovered site can only accept new transactions once the recovery procedure has finished. In [13] crashed sites can recover in parallel and at the same time several active replicas can serve them the needed data. The protocol in [14] proposes an adaptive approach which allows a recovering server to catch up with operational servers by transferring either the recent values of data items or the sequence of missed updates. In all these works, checkpoints are retrieved from operational replicas, and hence, at least one replica must be available to allow a database to recover. Our recovery architecture does not preclude replication, but does not rely on it for recovery; as presented earlier, Data Servers are not replicated.

Checkpoints often do not include all transactions executed in the system, since checkpointing and transaction processing run in parallel. Hence, a log tail with post checkpoint updates must be applied to get the database back to its consistent state before the crash. Much of the research community (e.g., [15], [16], [17]) and all of commercial database systems we are aware of take the approach of applying the entire log tail prior to taking new queries. We believe this should be avoided if possible in order to decrease the downtime of the database. On-Demand Recovery takes query traffic as soon as the database installs the last checkpoint, thereby amortizing the replay of the log tail across the incoming queries, increasing the availability of our system.

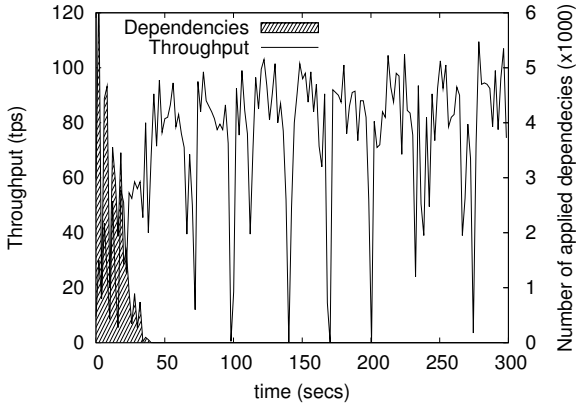


Figure 5. Throughput in scenario (i)

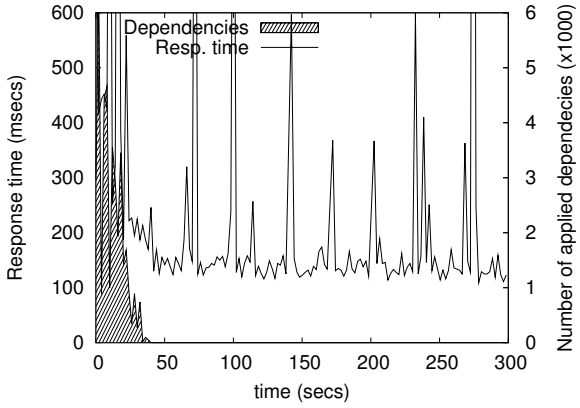


Figure 6. Response time in scenario (i)

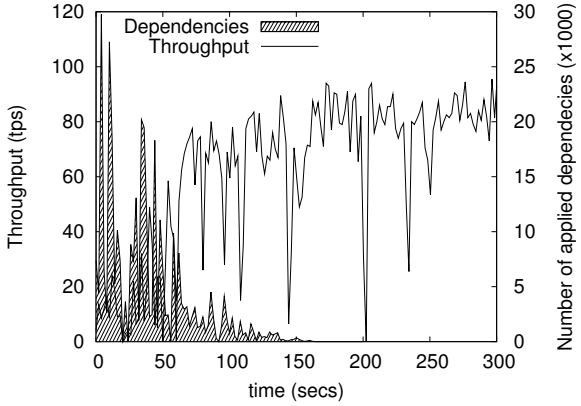


Figure 7. Throughput in scenario (iii)

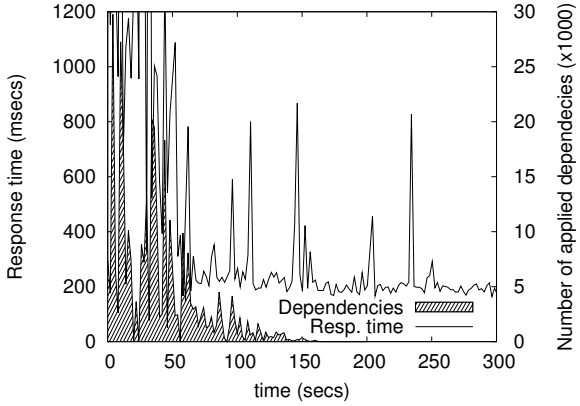


Figure 8. Response time in scenario (iii)

In [18], [19], the authors minimize the down time by accepting new queries and using them to indicate which items are to be applied from the log to the database: data items are recovered only after being accessed within a new transaction. The recovery process is done at page level. Likewise, On-Demand Recovery also identifies updates in the log that must be applied to the database before executing new queries. This is done at an operation level though, and is more appropriate for middleware solutions. Besides, all remaining operations are opportunistically applied, as background traffic, to speed up the execution of new queries.

The recovery scheme for in-memory databases of [20] has some similarities with our approach: redo logs are shipped to a database server to be stored, as done in our Durability Servers, and the database server keeps a stable version of the database for recovery purposes. In our approach, checkpoints and logs are kept in different locations. The checkpoints are used for On-Demand Recovery, and are not the only source of recovery information in the unlikely event of a crash of all databases. Differently, in [20] recovery information is stored only at the database server. Finally, the system in [20] is forced to wait until the server is recovered in the case of a failure.

In ARIES [17], [21], the authors introduce a recovery mechanism based on write-ahead logging and allow for some parallelization of recovery. In their scheme several passes are done over the log. The redo phase applies all the entries in the log tail, even if some of the transactions are to be undone. The undo log is analyzed to exclude aborted transactions from being re-executed, and a redo phase is used to apply the log tail on top of the latest checkpoint prior to restart. Differently, On-Demand Recovery eventually executes all

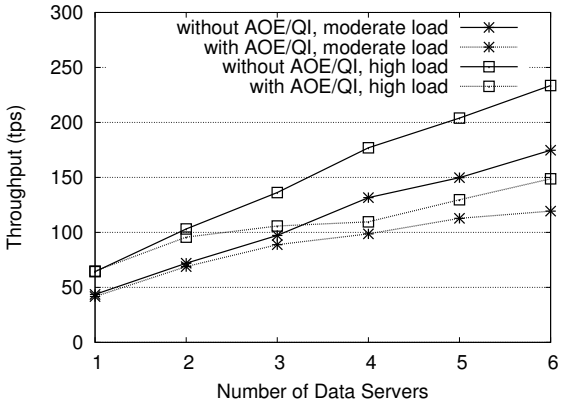


Figure 9. Data Servers per Recovery Server

redo entries; if a transaction has been committed to our Durability Server, it will be recovered. Also, in our scheme, we amortize the recovery of the log tail across the incoming queries.

There has been a significant body of work on the topic of optimal checkpointing [22], [23], [24]. These works have primarily focused on finding the optimal checkpoint interval. In [22] the authors show that the optimal checkpoint interval depends on the load of the system and hence is variable. In [23] the authors demostrate that the optimal checkpoint interval is directly related to the failure rate and in [24] the authors present a failure-dependent strategy, which under certain assumptions results in higher availability than previous methods. These works are complementary to our approach. On-Demand Recovery can be used to amortize the recovery of the remaining log tail on incoming queries.

VII. Conclusions

In recent years, some researchers in the database community [25], [2] have argued for a need to build application-specific solutions, which have been shown to outperform generic approaches. We support this view and argue that there are many gains possible, both in performance and availability, in middleware data management systems designed for specific application spaces (e.g., e-commerce) as compared to generic database solutions. In this paper, we presented a new recovery scheme that can speed up recovery by more than two times in a large application space represented by the TPC-C benchmark.

We argue that database systems should open up and make their internal parts more easily available to middleware developers, perhaps through APIs or novel frameworks (e.g., [26]), to facilitate application-specific innovations in distributed data management systems at fine-grain levels. Some work in the systems research community has already started moving in this direction, such as Stasis [25], a framework that provides low-level transactional support and many of the mechanisms typically found in databases.

Acknowledgment

The authors would like to thank the anonymous reviewers for their interesting comments and suggestions.

References

[1] M. Seltzer, "Beyond relational databases," *Commun. ACM*, vol. 51, no. 7, pp. 52–58, 2008.

[2] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (it's time for a complete rewrite)," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 1150–1160.

[3] P. Bernstein, M. Brodie, S. Ceri, D. Dewitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman, "The Asilomar report on database research," *SIGMOD Record*, vol. 27, no. 4, 1998.

[4] L. Camargos, F. Pedone, and M. Wieloch, "Sprint: a middleware for high-performance transaction processing," in *Proceedings of the 2nd EuroSys European Conference on Computer Systems*, 2007, pp. 385–398.

[5] J. N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[6] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[7] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[8] V. Zuikeviciute, "On non-intrusive workload-aware replication," PhD thesis, University of Lugano, 2009.

[9] E. L. Cashin, "ATA over Ethernet: putting hard drives on the lan," *Linux J.*, vol. 2005, no. 134, p. 10, 2005.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 29–43.

[12] B. Kemme, A. Bartoli, and O. Babaoglu, "Online reconfiguration in replicated databases based on group communication," in *Proc. of the International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2001, pp. 117–130.

[13] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso, "Non-intrusive, parallel recovery of replicated data," in *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, 2002, p. 150.

[14] W. Liang and B. Kemme, "Online recovery in cluster databases," in *Proceedings of the 11th international conference on Extending database technology*, 2008, pp. 121–132.

[15] E. Lau and S. Madden, "An integrated approach to recovery and high availability in an updatable, distributed data warehouse," in *Proceedings of the 32nd international conference on Very large data bases*, 2006, pp. 703–714.

[16] S.-O. Hvasshovd, Ø. Torbjørnsen, S. E. Bratsberg, and P. Holager, "The Clustra telecom database: High availability, high throughput, and real-time response," in *Proceedings of the 21th International Conference on Very Large Data Bases*, 1995, pp. 469–477.

[17] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, 1992.

[18] R. L. Rappaport, "File structure design to facilitate on-line instantaneous updating," in *Proceedings of the 1975 ACM SIGMOD international conference on Management of data*, 1975, pp. 1–14.

[19] E. Levy and A. Silberschatz, "Incremental recovery in main memory database systems," *IEEE Trans. on Knowl. and Data Eng.*, vol. 4, no. 6, pp. 529–540, 1992.

[20] R. Rastogi, P. Bohannon, J. Parker, A. Silberschatz, S. Seshadri, and S. Sudarshan, "Distributed multi-level recovery in main-memory databases," *Distrib. Parallel Databases*, vol. 6, no. 1, pp. 41–71, 1998.

[21] C. Mohan and H. Pirahesh, "ARIES-RRH: Restricted repeating of history in the aries transaction recovery method," in *Proceedings of the Seventh International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1991, pp. 718–727.

[22] E. Gelenbe, "On the optimum checkpoint interval," *J. ACM*, vol. 26, no. 2, pp. 259–270, 1979.

[23] Y. Ling, J. Mi, and X. Lin, "A variational calculus approach to optimal checkpoint placement," *IEEE Trans. Comput.*, vol. 50, no. 7, pp. 699–708, 2001.

[24] A. N. Tantawi and M. Ruschitzka, "Performance analysis of checkpointing strategies," *ACM Trans. Comput. Syst.*, vol. 2, no. 2, pp. 123–144, 1984.

[25] R. Sears and E. Brewer, "Stasis: flexible transactional storage," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 29–44.

[26] Http://gorda.di.uminho.pt/.

APPENDIX: ON-DEMAND RECOVERY CORRECTNESS

In the following, C is a database checkpoint, and QI and LT are, respectively, the corresponding Query Index and log tail received by the new Data Server. VTI is the vector of transaction identifiers associated to C or QI.

Proposition 1: Recovery avoids lost transactions.
Proof sketch: For a contradiction, assume T is lost upon recovery. From the database properties, T is not included in any C. Since T committed, however, it must be in the Durability Servers' logs, but T is neither in QI nor in LT—otherwise it would not be lost. In the former case, it must be that T precedes some transaction in VTI_{QI}, and thus, T must be in QI, unless it was removed from the index after being included in it. But T is only removed from QI after it is included in a C, a contradiction. □

Proposition 2: Recovery avoids duplicated transactions.
Proof sketch: For a contradiction, assume T is executed more than once. Thus, T must be in more than one of C, QI, and LT. From the algorithm, the invariant next holds: $VTI_C[t] \leq VTI_{QI}[t] \leq T$, where t is the working task that executed T and $a \leq b$ means t executed a before b, or they are the same transaction. From the algorithm, (a) only transactions in QI that succeed transactions in VTI_C are executed, and (b) only transactions that succeed transactions in VTI_{QI} are included in LT. Thus, if $T \in C$, it will not be executed even if $T \in QI$ (from (a)) and $T \notin LT$ (from the invariant and (b)); if $T \in QI$, then $T \notin LT$ (from (b)), a contradiction that completes the proof. □

Proposition 3: Recovery preserves commit order of conflicting transactions.
Proof sketch: Let T commit before T' before recovery. We consider first three cases: (a) $\{T, T'\} \subseteq C$; (b) $T \in C$ and $T' \in QI$; (c) $T \in QI$ and $T' \in LT$. Case (a) is trivial; cases (b) and (c) hold because after installing C, recovery executes first transactions in QI and then in LT. Notice that if $T' \in C$ then $T \in C$, from database concurrency control and recovery. The remaining cases are: (d) $\{T, T'\} \subseteq QI$, (e) $\{T, T'\} \subseteq LT$, and (f) $T \in LT$, $T \notin QI$, and $T' \in QI$. In case (d), since T committed before T', it will be included in QI before T'; moreover, since they conflict, T's updates will precede T''s in QI, determining their execution order upon recovery. In case (e), T will enter the Durability Server logs before T'. Since transactions in LT are replayed following their order in LT, T's updates will precede T''s. Finally, in case (f) notice that since T committed before T' and $T \notin QI$, then T must have been later removed from QI. Only transactions included in a checkpoint are removed from QI, thus $T \in C$, and we fall back to case (b)—additionally, from Proposition 2, even if $T \in LT$, it will not be executed again, upon recovery, after T'. □