Database Recovery Mechanisms : A Survey

Midhul Varma
IIT Guwahati
midhul@iitg.ernet.in

Preetham Kamidi
IIT Guwahati
kamidi@iitg.ernet.in

ABSTRACT

It is essential for database management systems to ensure reliability of the data which they store even in case of failure. For this purpose implementations of well studied recovery techniques are built into these systems. In this survey, we start off by describing the basic concepts related to database recovery and then discuss some of the popular recovery mechanisms used in traditional on disk DBMS. Following this, we explore recovery in the context of modern database systems such as Main Memory and Distributed databases. Each of these settings poses its own set of challenges when it comes to recovery. We attempt to enumerate these challenges and discuss some of the solutions implemented in existing database systems.

Keywords

Databases, Database Management Systems, Distributed Systems, Distributed Databases, Main Memory Databases, Recovery, Crash Recovery, Survey

1. INTRODUCTION

Databases form a crucial part of systems everywhere, important information is stored, edited and used in wide array of scenarios. Ranging from traditional standalone databases to distributed databases, information stored in databases is of great importance. In the event of any kind of failure in such systems, database recovery becomes a major issue. It involves restoring the state of database to a consistent state in the event of a system failure. Recovery might involve several procedures depending upon the type of failure that occurred. Section 2 describes some important basic concepts and terminology which is prerequisite to understanding recovery. Section 3 details recovery mechanisms in traditional on-disk database systems. Section 4 deals with recovery techniques in modern database systems and Section 5 concludes the survey.

2. BASIC CONCEPTS

2.1 Transactions

A transaction is a sequence of logical operations performed on the database constituting a unit phase of work. For example, transferring money from one bank account to another is a single transaction but involves several modifications to tables of both user accounts in the bank's database. Every transaction must ensure that it leaves the database in consistent state after execution. It is ideal for database systems to satisfy the famous ACID properties (Atomicity, Consistency, Isolation & Durability). Recovery mechanisms play an essential role in ensuring the Atomicity and Durability guarantees:

1. **Atomicity**: While a transaction is executing it could make several changes to data items in the database. In the event of a failure, the database should rollback the partial changes made to it. In short it must be either all of the changes or none of the changes.

2. **Durability**: After a transaction has successfully executed, it *commits*. Once a commit is processed, the transaction cannot be rolled back in any case. If the system fails or shuts down, it must be ensured that the changes made by all committed transactions stay persistent in the database.

2.2 Types of Failures

Failures in databases can be of broadly three types:

1. **Transaction failure**: Transactions can be aborted by the database management system due to several reasons. Sometimes the transaction might fail due to lack of resources at that particular point of time, for example the resource required might be in a deadlock or not available at all. Other transaction errors include bad inputs which cause abort by the transaction itself. Transaction errors in a typical database system occur once in a few minutes.

2. **System failure**: System failures include operating system level failures such as bus errors. Main memory content loss is also a system failure but this might be caused by power outage. System failures may occur once in a few months depending upon the situation.

3. **Disk failure**: Disk failures are when the non-volatile storage gets corrupted due to the formation of bad sectors etc. Such kind of errors are not so frequent as transaction errors, but eventually occur after sometime usually once in a few years.

2.3 Recovery Manager

Recovery Manager (Figure 1) is one of the most important part of a database management system. It is also one of the toughest one to implement as there are large number inconsistent states of a database during a failure and the recovery manager has to be successful in restoring it to a consistent state every single time. The recovery manager is

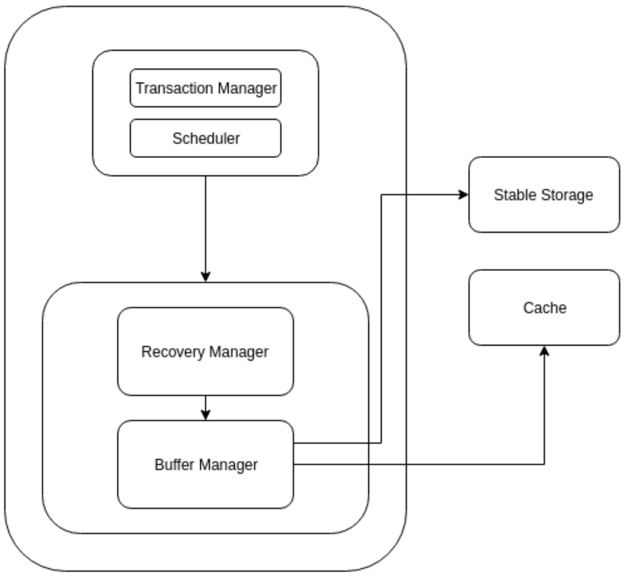


Figure 1: Modules of DBMS

responsible for storing auxiliary information while processing transactions. It has to ensure atomicity and durability of transactions while restoring the state of the database after restarting from crashes. It closely interacts with the buffer manager to achieve some of its functionality.

Different types of database management systems employ different techniques to solve recovery problems. We will look into each one of them in detail in the following sections.

3. TRADITIONAL ON-DISK DBMS

In this section we discuss various recovery mechanisms for traditional centralized on-disk Database Systems and how they have evolved over time. In the process, we try to build a list of characteristics based on which recovery mechanisms can be compared and contrasted. We focus on mechanisms which ensure recovery to a consistent state after system failures, and hence assume that data in secondary storage elements is still intact.

3.1 The Problem

Consider a database system which is running on a single computer and is processing transactions from a client. While processing these transactions it needs to read/write data from secondary storage which in this case is a disk. Each of the transactions individually satisfies the consistency properties and hence after each transaction successfully completes the state of the database will surely be in a consistent state. Now if the system experiences a sudden failure while in the middle of processing some transactions(s), the state of data on disk may no longer be consistent. Since, after the failure all volatile memory is lost, upon restart of the system the only information which the DBMS has access to is the data which was written onto disk prior to the failure. Looking just at the present state of the disk, the DBMS cannot deduce which transactions were running during the failure and cannot take any steps for recovery. Clearly, apart from simply reading and writing data items to/from the disk, the DBMS also needs to keep track of some auxiliary information to enable recovery in case of such failure. What extra information it stores, how it stores this information and how it uses this information for recovery after restart are essential questions which any recovery mechanism must answer.

Interaction with the disk is usually done in units of blocks, and writing/reading entire disk blocks for every data item which is read/written is inefficient. To remedy this database systems cache disk blocks into main memory using a buffer manager and delay writes to disk in a lazy manner. These delayed writes create problems during recovery, as there is no guarantee that all the changes made by transactions which have finished execution have been written to disk. Hence, the functionality of the recovery manager is dependant to that of the buffer manager.[13] gives a convenient classification of the behaviour of buffer managers:

- **Steal approach**: Changes made by a transaction to a data item can be written to disk before the transaction commits.
- **Force approach**: When a transaction commits all changes are immediately forced to disk.

Implementing recovery for a buffer manager which uses a no-steal, force approach is trivial, as at any moment of time only the changes made by committed transactions are seen on disk.

3.2 Shadow Paging

Some of the earliest literature about database recovery mechanisms dates back to 1977 in an article by Raymond Lorie[10]. It describes the design and implementation of a simple database system which stores data in the form of segments (similar to files). Each segment is allocated pages which are mapped to slots on disk and the mapping of slots allocated to each segment is tracked using a directory table. When a page needs to be read it is simply fetched into the buffer pool from the mapped slot. In contrast, when a modified page needs to be written to disk, the mapped slot on disk is not overwritten. Instead, an alternate new slot is allocated and the modified page is written to this new slot. This creates a dual mapping for the modified page and the old slot still contains a copy of the page before it was modified, sort of acting like a ”shadow”. Further modifications to the page are written to the new slot and the old shadow slot is left untouched. This continues, until a special SAVE command is executed on the segment. Upon execution of the SAVE command, the new slots of all the modified pages are finalized and the shadow slots can then be removed. If a failure occurs between two successive SAVEs of a segment, since the shadow slots still contain consistent snapshots of the pages, consistent state can easily be restored. The system also takes certain steps to ensure problem free storage in case of failure during execution of the SAVE command. The recovery manager of IBM's famous System R was designed to use a similar mechanism and is detailed in [8]. This approach to recovery enables very rapid recovery from failures but suffers from a number of drawbacks:

- The overhead associated with storing shadow pages is very high, as entire pages need to be cloned even for modifications on small data items.
- The mechanism does not extend easily to support concurrent execution of transactions as a SAVE command finalizes the present state of every page which could contain modifications made by several transactions running at that moment, some of which may not have yet committed.

The latter point is a major drawback for these kinds of techniques because a modern DBMS has to at the very least support concurrent transactions to be of any practical purpose.

3.3 Write Ahead Logging(WAL)

The shortcomings of shadow paging motivate the need to support in place updates of data items, while tracking information per transaction instead of per page or per file. Write Ahead Logging (WAL) is a technique which has these features. The basic principle is to maintain a separate log on stable storage which tracks the history of write operations performed by transactions while they are executing. This log is an append only structure and new log entries are added to its end. Each log entry stores the id of the transaction responsible for the write along with some other information about the write for recovery. Operations like Commit and Abort are also usually logged. The exact information stored varies from algorithm to algorithm. Any WAL based recovery algorithm must satisfy the following two properties:

- The log entry corresponding to a write MUST be written to disk before the actual write is affected on disk.
- Before a transaction is committed, all of its actions in the log must be on disk (i.e the log needs to be force written while committing a transaction).

The first property helps ensure atomicity, as all writes on disk are surely present in the log. The latter point helps ensure durability, since when a transaction is committed there is a guarantee that it's actions have at least been recorded in the log.

The way in which recovery manager uses information in the log to restore the system to a consistent state after system restart from failure depends on behaviour of the buffer manager:

- **No force approach**: Writes of committed transactions may not have been flushed to disk and hence, committed transactions may require Redo.
- **Steal approach**: Writes of uncommitted transactions may have already been flushed to disk and hence, may require Undo.

Hence depending on buffer manager behaviour, recovery algorithms may require both undo/redo, undo only, redo only, neither undo/redo. [1] presents simple algorithms for each of these cases and discusses their pros and cons. Undo + Redo algorithms are the most complex but give the buffer manager more flexibility, allowing the database systems processing throughput to increase. The exact details of how the redos and undos are performed depends on the type of information stored in the log records.

Each log entry essentially records an operation performed by a transaction on a data item. The information stored to describe each operation can be done at different levels:

- **Physical Logging (low level)**: Changes are stored at byte level.
  - Example: Transaction T1 changed 4 bytes on page #301 at offset 0x43 from <before image> to <after image>.
- **Logical Logging (high level)**: Abstract operations and the arguments passed to them are recorded.
  - Example: Transaction T1 inserted node with key value 37 into B-Tree rooted at <root-node>.

These different levels of logging present a trade-off in several aspects:

- Physical log records tend to consume more space.
  For example: Inserting a record into a page might cause the entries below the insertion point to shift and hence, it appears as if the entire page was modified. Logical log records on the other hand are more compact and can express the modification of large set of bytes within a single small record.
- Undo/Redo operations on physical log records are very easy to perform. To undo, the recovery manager simply needs to replace the specified range of bytes with the <before-image> and to redo it simply needs to replace them with the <after-image>. In the case of logical logging, Undos and Redos require much more intelligence to embedded into the recovery manager as it may need to be aware of the semantics of the associated operations.
- Redo operations on physical log records are idempotent.
  For example: Consider the log record "bytes 100-102 of page #305 set to 0x56 0x78 0x48". On a given page, no matter how many times this operation is done, the outcome will be the same. Hence, during restart the recovery manager can blindly redo the operations of all committed transactions. On the other hand if we consider a logical record of the form "record with key 88 and value 40 is inserted into B-tree x". Executing this operation twice is not the same as executing it once. In this case the recovery manager cannot blindly redo all operations of the committed transactions, as the effects of some of these operations may have already been flushed to stable storage. To support logical logging, the recovery manager needs to maintain information about which operations of transaction have already been written to disk. This is usually achieved

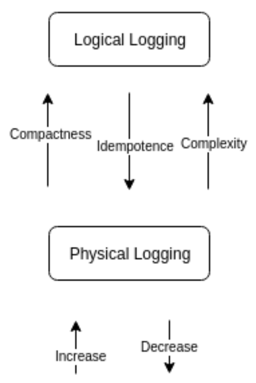


Figure 2: Physical and Logical Logging

by assigning unique identifiers to log records(Log Sequence Numbers or LSNs) and maintaining the LSN of the oldest log record whose corresponding update has not yet been written to disk for each active transaction.

In summary the Figure 2 illustrates the trade-offs between physical and logical logging.

Physical and Logical logging are not discrete. It is possible to construct schemes which lie somewhere in the middle of this classification. One such example is **Physiological Logging** which attempts to harness the best of both methods. In this logging scheme updates are physical at the page level but can be logical within the page.

- Example: "Insert record X into Page #503"

Another important point to note is that before Undo and Redo operations of logical log records are performed, the database needs to be brought to the same state as it would have been in case there was no failure.

Write Ahead Logging (WAL) allows per transaction, finer granularity logging and hence supports concurrent execution of transactions unlike Shadow Paging where the SAVE operations were always at the granularity of segments (group of pages). Hence the granularity of logging affects the granularity of locking which determines the allowed level of concurrency.

To avoid logs in WAL based recovery mechanisms from indefinitely growing, a check-pointing mechanism needs to be implemented. Check-pointing periodically synchronizes the state of main memory and stable storage, and allows garbage collection of old entries.

From the above analysis of recovery mechanisms we note the following aspects which determine their performance and efficiency:

- **Space overhead**: Amount of additional space in stable storage needed for recovery purposes.
- **Flexibility given to buffer manager**: Whether or not no-steal/force approach is required. Support for steal and no-force is considered to offer highest flexibility.
- **Concurrency Support**: This is determined by the granularity of locking allowed by the recovery mechanism (file level, page level, record level etc.). The finer the granularity of locking the higher the allowed concurrency.
- **Check-pointing efficiency**: Check-pointing can be a time consuming process and might put a bottleneck on the achievable throughput. Hence efficient check-pointing is very important.

3.4 ARIES

In the 1980's there were several proposals of WAL based recovery mechanisms each of which made different assumptions about the behaviour of buffer managers.In 1992 Mohan et al. published a paper titled Algorithms for Recovery and Isolation Exploiting Semantics [11].
ARIES operates according to three fundamental principles:

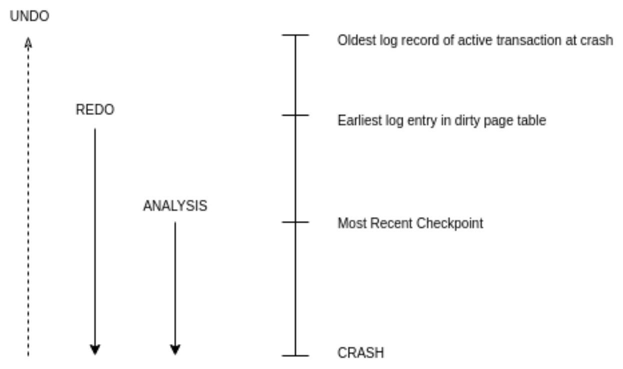
1. **WAL**: Write Ahead Logging (WAL) of operations by transactions.
2. **Repeating History**: After restart, ARIES repeats all the operations performed by transactions prior to crash including those which have not committed. It then undoes the operations of the uncommitted transactions in reverse order.
3. **Logging changes during Undo**: The process of undoing uncommitted transactions is itself logged to deal with the case of failure during recovery.

The operation of ARIES can be vaguely described in 3 phases as given in [13]:

- **Analysis Phase**: It determines the point in the log at which to start the next Redo phase, a conservative superset of the pages in the buffer pool that were dirty at the time of crash and a list of transactions that were active at the time of crash.
- **Redo Phase**: Redoes the operations of all transactions which corresponding to pages in the dirty page table which was constructed in the previous phase.
- **Undo Phase**: Operations of uncommitted transactions which are in the active transaction list are undone in reverse order.

The exact implementation details of ARIES are beyond the scope of this article, and we conclude by discussing some of the key unique features of ARIES which may be the reasons for its widespread adoption:

- *Flexible buffer management*- ARIES assumes a steal, no-force approach.
- *Fine granularity locking*- The redo phase brings the database back to the state just before crashing. This restoration of state allows the usage of physiological logging.



UNDO
REDO
ANALYSIS
Oldest log record of active transaction at crash
Earliest log entry in dirty page table
Most Recent Checkpoint
CRASH

Figure 3: Three Phases of Restart in ARIES

- *Efficient check-pointing*
- *Support for partial rollback*- Instead of fully undoing uncommitted transactions, ARIES supports partial undo.
- *Simplicity*- Several of the previous WAL based mechanisms which offered similar features were much more complex. Its relative simplicity is a major plus point.
- Takes steps to ensure recovery even in the case of repeated failures(Failure during restart).

4. MODERN DBMS

4.1 Main memory databases

To understand recovery related challenges in Main Memory Database systems, one must first have a basic understanding of the system model of Main Memory databases and how the recovery concepts discussed in previous sections are relevant to this model.

A Main Memory Database System (MMDB) or In-memory Database System (IMDB) is a type of Database Management System which relies primarily on Main Memory for storage of it's data. Unlike the on disk database systems discussed in the previous section which store the primary copy of the database on disk and cache blocks of it in Main Memory to increase throughput, MMDBs typically store the primary copy of the database in main memory itself. Atomicity, Consistency and Isolation guarantees can be implemented in the same way as normal on disk DBMS, except disk I/O is no longer necessary for reading and writing data items. The volatility of main memory brings the Durability offered by MMDBs into question. One way to ensure partial durability is to periodically take backups of consistent snapshots of the database onto stable storage. This does not offer full durability as failure in the middle of two successive backups could lead to loss of data. To ensure full durability MMDBs also need to use logging techniques similar to those used by on-disk DBMS in conjunction with periodic backups.

We now describe a basic system model to continue the discussion:

- A single centralized system which stores data in Main Memory and is continuously processing transactions from a client.

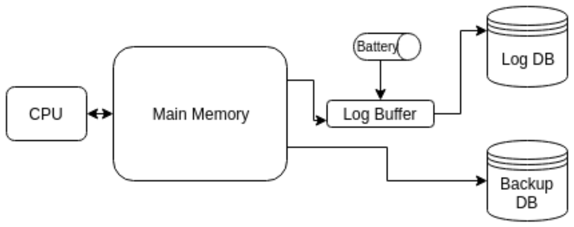


Figure 4: Main memory DBMS model

- An Archived/Backup version of the database in consistent state stored on reliable secondary storage like a disk or RAID array.
- A log to which changes made to the database in memory are written to for the purpose of recovery. The log needs to be on non-volatile storage, but writing log entries directly to disk requires I/O and that kills the purpose of storing all the data in Main Memory. Hence, log buffers are stored in separate main memory which is supported by backup battery power supply making it effectively non-volatile. The buffers can then be asynchronously flushed to disk without causing much damage to system throughput. Note that this arrangement is feasible because the size of the buffers does not need to be as large as the size of the primary main memory store itself.

The recovery mechanism used in this model is a simple WAL (Write Ahead Logging) protocol similar to that described in the previous section. Changes to data items in main memory are first logged into the log buffers. When a transaction commits a corresponding log record is written to the log buffers. Since the log buffers are non-volatile we do not need to wait for the buffers to get flushed onto disk before processing the commit. This eliminates the need for I/O while finalizing commits. Frequent checkpointing is also performed to keep the main memory, archived copies and log synchronized and enable garbage collection of old log entries.

We now revisit the different types of failures and analyze their effect on the above described model:

- **Transaction failures**: Just like in on-disk DBMS, changes made by transactions which get aborted during execution must be undone in Main Memory. The undo process may also need to be logged by writing records onto the log buffers.
- **System Failure**: Unlike in the case of on-disk DBMS, upon system failure, the primary database copy which is stored in main memory is lost. Hence, after restart the entire database must be reconstructed. To do so first the latest archived snapshot is loaded from the backup disk into main memory. Log buffers are flushed onto disk and then the log is read, analyzed and used to perform recovery on top of the backup snapshot. ARIES like recovery algorithms modified for main memory settings can be used to recover from the log.

- **Media failure**: Disk failure can cause loss/corruption of data in backup copies as well as the log. The problem is partially alleviated by making use of redundancy. Since secondary storage is far cheaper compared to main memory, maintaining multiple copies of data on different disks is feasible.

In this kind of model, some of the aspects which could be potential bottlenecks to system performance are listed below:

- *Number of log records written per transaction-* Since every change to data in memory requires writing log records to the log buffer, if the average number of records written per transaction is high, this could lead to significant overhead due to logging. To reduce the number of log records written, logical logging is a good choice, but introduces complexities during checkpointing and recovery after restart.
- *Checkpointing efficiency-* Since checkpointing is performed frequently in the system, it is essential that it does not block the processing of normal transactions.
- *Recovery Time-* Unlike in on-disk DBMS, restart after failure requires reloading the entire database contents from disk and then performing log based recovery. It could take a very long time for this long recovery process to complete making it a problem for applications which require high availability.

Beginning from the 1980's itself there has been extensive research into the design and implementation of Main Memory Database systems and their associated recovery mechanisms. One can find several papers in literature dedicated to this topic [4] ,[5] ,[6], [9]. Diving into the details of these papers would be the topic of an independent survey on it's own. Instead we move on and explore some modern systems and how they implement recovery mechanisms.

4.2 RAMCloud

RAMCloud is a distributed DRAM-based storage system which stores data fully in DRAM, and keeps backup replicas in disks/flash storage. It tries to achieve extremely low latencies (in order of microseconds) and maintain high availability. Maintaining in-memory replicas of data is extremely costly and is hence only a single copy of data is maintained in main memory at any given time and multiple backup replicas of data are maintained in disks at different nodes, as this arrangement is much cheaper.

When a node in such a system crashes, the data which that node is responsible for is no longer present in the main memory of any node in the system. To assure availability, it needs to recover quickly from crashes i.e the data which the crashed node is responsible for must be transferred from backup replicas on disk into the main memory of other live nodes in the system so that is available for reads again. For this purpose the authors of [12] propose a recovery mechanism inspired from log structured file systems, which exploits the scale of the system in speeding up the recovery process. To understand how this recovery process works one needs a basic understanding of RAMCloud's system architecture.

A RAMCloud system is essentially a cluster of storage servers connected through a high speed network. (The paper assumes usage of Infiniband NICs so that network communication is not the bottleneck). Each storage server consists of two components:

- *Master* - Stores RAMCloud data in it's DRAM and services client requests.
- *Backup* - Stores backup replicas of data of other masters on server's disk.

The data of a master is never stored on the backup of the same server, because it is likely both will become unavailable during failure. Instead of backups for storing, replicas are chosen intelligently so that they are not even on the same rack as the master. Data is stored in the form of simple key value pairs. Each key value pair also has a version number associated with it. There is one special server called the coordinator, which is mainly for storing configuration information and does not take part in most client requests. The coordinator decides what partition of the data (set of keys) each server should store. When a client makes it's first request for an object with a particular key, the request goes to the coordinator which responds with the corresponding mapping. The mapping of partitions to servers is cached in each client so that future requests don't need to go to the coordinator. The data in each master's DRAM is stored in the form of Log Structured Storage. This is essentially an append only log which contains the history of modifications made to key value pairs. Whenever the value of an object with a specific key is modified, a new log entry containing the key, new value and incremented version number is appended to the end of the log. Each master also maintains a hash table structure which stores pointers to the latest version of each data item in the log. Writes to the log in main memory are written to all of the chosen backups. To avoid the need for I/O while processing write requests, backups store incoming log entries in non-volatile main memory buffers and flush them to disk eventually in large batches. To ensure non-volatility of backup buffers, they are backed up by battery power supply similar to what was discussed in the simple MMDB model previously. The main memory log is split into units of segments, which are garbage collected by a periodically running cleanup process. As pointed out in the paper there are several advantages of using this kind of log-structures storage:

- Unified structure of data in both main memory and backups. Masters can use a single mechanism to manage data in both in memory and on backups.
- Provides an efficient and simple memory management mechanism.

When a server fails, the data stored in master's main memory is no longer available in the system. To recover, the replicated log segments which are stored in other backups must be read into main memory and processed to find

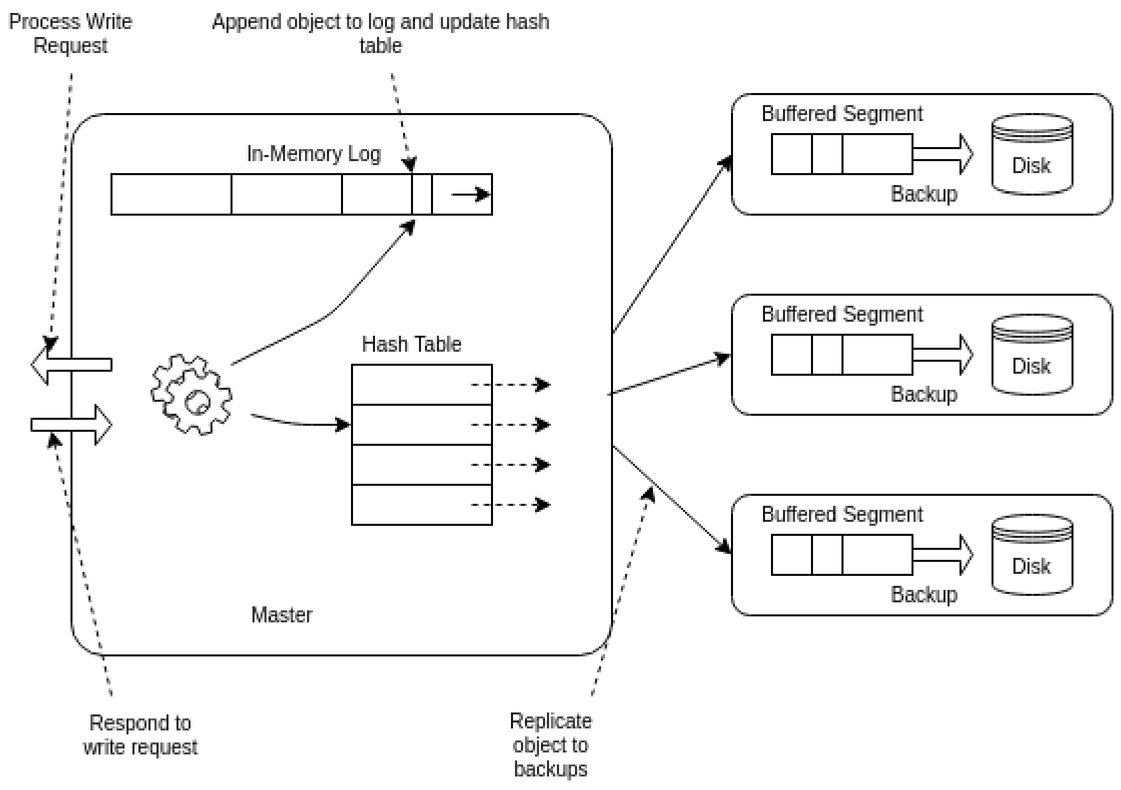


Figure 5: RAMCloud System Architecture

the latest version of each object. The replicated log on disk could be huge and reading it from a single backup into a single master is bounded by the disk bandwidth and could take several minutes to complete. To speed up this process, RAMCloud exploits the scale of the system by reading replicated segments from multiple backups into multiple different masters and processing them in parallel. The following are the steps which it takes:

- **Setup**: The coordinator first finds the backups where the replicated segments of the master that crashed are stored. It then selects a group of online masters and allocates a partition of the crashed master's data set to each of these. The backups are then notified about which master is responsible for the recovery of which partition.
- **Replay**: The backups start reading segments from disk and transfer the records in each segment to the master responsible for recovery of it's corresponding partition. The masters on receiving records add them to their own logs essentially taking responsibility of the corresponding data objects.
- **Cleanup**: Once the replay is completed, the data of the master which crashed is distributed to other online masters and is available to read requests. The backup logs of the crashed master can then be cleaned up.

Results in the paper claim that with 100 recovery masters running in parallel fetching replicated segments from 1000 backup disks over a 10 Gbps network, 64 GB of data can be recovered within 1 second, making it highly available.

[14] and [2] also use similar ideas of log structured storage to enable easy and fast recovery from failures.

4.3 Distributed Databases

Distributed DBMS faces much more complicated problems than a traditional centralised DBMS in terms of recovery mainly due to the following reasons:

- Networking failures now play a major role in a transaction as failure of a communication link might occur in the middle of a transaction.
- System failure at a remote site where the subtransaction needs to execute affects the whole transaction.

Despite such problems, the recovery manager of a distributed DBMS must guarantee atomicity even if combination of problems occur at the same time i.e. either all the subtransactions need to commit or none of them commits. To deal with these kinds of problems, distributed DBMS use commit protocols to achieve consistency. Variants of commit protocols with their advantages as well as tradeoffs are briefly described below.

4.3.1 *Two-Phase Commit Protocol(2PC)*

In a distributed DBMS, the transaction manager where the transaction originates is called a coordinator and the transaction managers where subtransactions execute are called subordinates. A transaction manager can be the coordinator for a transaction and a subordinate for some other transaction. Now let's look into two-phase commit protocol(2PC):

- When a user wants to commit a transaction, commit message is sent to the coordinator. The coordinator now sends a prepare message to the subordinates.
- Each subordinate after receiving a prepare message force-writes a prepare log or an abort log depending upon its decision to abort or commit the subtransaction. After force-writing to its log, it sends a no or yes message to the coordinator.
- The coordinator after sending a prepare message to its subordinates, waits for yes/no messages.If each one of its subordinates sends a yes message, the coordinator sends a commit message after force-writing to its log record. If any one of the subordinates sends a no message the coordinator force-writes an abort message to its log and sends back an abort message to all of them.
- If the subordinate now receives an abort message, it force-writes an abort log record, aborts the transaction after sending an ack message to the coordinator. If the subordinate now receives a commit message, it force-writes a commit log record, commits the subtransaction after sending an ack message to the coordinator.
- The coordinator, after receiving ack messages from all the subordinates writes an end log record for that particular transaction.

4.3.2 *Recovery using 2PC*

Recovery manager after a crash reads all the logs and processes the transactions at the time of crash. Depending on the scenario, recovery method is applied using 2PC procedures. Some of the scenarios:

- For a transaction T, if there exists a commit or abort log record we need to redo or undo the transaction. If this transaction manager is the coordinator, it needs to send a commit or abort messages to all its subordinates until it receives ack message from all of them. This is just in case to check if there are other failures in subordinates. After receiving ack messages from all the subordinates the coordinator writes end log record for the transaction T.
- If only a prepare log exists but no commit or abort log, we must contact the coordinator repeatedly to know the status of the transaction T. The coordinator then responds with a commit or abort message to which the subordinate force-writes a commit or abort log depending on the message received and undos or redos the transaction T. After this the subordinate sends back an ack message to the coordinator and end log record is written.
- If there is no prepare, commit or abort record for T we can abort and undo T as it hasn't been voted to commit before the crash.

Drawbacks:

- Major disadvantage of 2PC is blocking. If the coordinator fails for a transaction T, the subordinates which have votes yes are not able to decide if they can commit or abort until the coordinator comes back online. All the resources held by that subordinate are blocked. This might lead to a situation where the resources are held forever by the subordinate.

4.3.3 *Three-Phase Commit Protocol(3PC)*

To overcome the problem of blocking, three-phase commit (3PC) employs one more phase. In this protocol, the coordinator sends prepare messages and receives yes/no messages just like 2PC. But after this the coordinator sends out a precommit message to all of its subordinates instead of a direct commit message. When the coordinator receives sufficient number of acks it force-writes a commit log record and now sends out a commit message to the subordinates. The main advantage of this commit protocol is the coordinator effectively postpones the commit message until it is sure that each one of its subordinates is ready for commit. If at all the coordinator fails, the subordinates can talk with one another checking if any one of them received a precommit message, hence eliminating the possibility of a blocking scenario.

4.4 Replicated databases

To achieve durability and low service latencies most modern distributed database systems replicate data on multiple nodes in a network. Clients can then contact nodes nearest to them for transaction processing and hence experience lower latency. The important question which leads to a classification of these types of systems is how consistency is achieved among the replicas:

- **Strong consistency**: If a transaction makes changes to data items, it must wait for all replicas to acknowledge these changes before it commits. This means when a transaction successfully commits, the changes made by it are guaranteed to be available on all replicas.
- **Weak consistency**: Changes made by a transaction are affected locally on the target replica and are asynchronously communicated to the other replicas. The only guarantee is that the system will eventually reach a consistent state.

The recovery protocols used in these systems are strongly coupled with the replication protocols themselves. [7] provides a comprehensive survey of a broad range of recovery protocols designed for replicated databases. The paper classifies recovery protocols on the basis of the following characteristics:

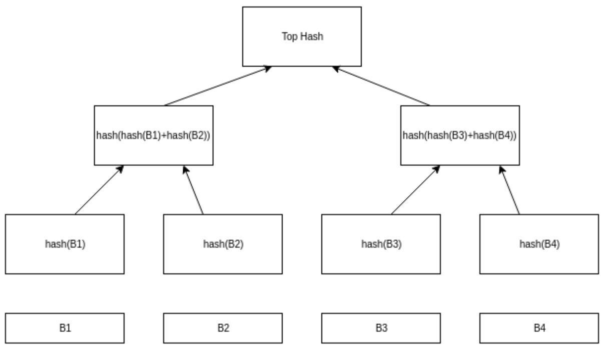


Figure 6: Merkle Tree

- *Transfer model*- After a node is brought back up after failure, it must obtain the present state of the database from replicas. This can be done in several ways such as, transferring the entire database, version based incremental transfer or resending of lost messages in network.
- *Concurrency control during recovery*
- *Work distribution*: During recovery whether the processing is centralized or distributed among multiple nodes. RAMCloud discussed previously is an example which distributes work during recovery.

4.5 Amazon's Dynamo

Dynamo is Amazon's highly available key value storage system which powers some of the company's core online services. It is targeted at systems which can allow compromise in consistency for the sake of higher availability. Just like RAMCloud, Dynamo also offers storage of objects in the form of simple (key, value) pairs. [3] presents a detailed discussion of implementation of the Dynamo system. It replicates data among several nodes and achieves weak consistency using an optimistic replication protocol. When nodes in the system undergo system failure, it may take a considerable amount of time to restart them, because first the failure needs to be detected by the membership protocol and then may require human intervention to fix. In the time during which the node is offline, it misses all updates which happen at other replicas. In systems which are serving millions of requests every second the failed node may fall considerably behind the other replicas. To recover after restart, the failed node must synchronize with other replicas as quickly as possible without consuming too much of the network bandwidth so that normal operation is not affected.

For handling permanent failures, Dynamo uses an anti entropy protocol*(gossip)* along with special data structures called *merkle trees* to quickly synchronize replicas after restart from failure.

To understand the concept behind merkle trees we consider the example of how two nodes on a network can synchronize a single long file of data efficiently. Consider two nodes A and B, both of which hold copies of a particular data file. Let's say A has been offline for some time and wants to synchronize it's copy of the data file with B. Both the nodes maintain a special tree structure in addition to the data blocks of the file. The lowest level of nodes in the tree contain cryptographic hashes (such as MD5 or SHA1) of the content of blocks of the data file. The next higher level of nodes contain the hashes of the content of their children nodes, and the pattern continues upto the root node. Now for synchronization the nodes don't have to exchange the entire file. A starts off by asking B for the hash of its root node. If the hash of B's root node is the same as that of A's root node, it means the copies of file on A and B are exactly the same, so no more messages need be exchanged over the network. If the hash is not the same, the hashes of children of B's root are requested by A. The process then repeats recursively until the the data blocks of the file which are unmatched are recognized. A then asks B to transfer these blocks and then replaces the corresponding blocks in it's copy of the data file. Hence using this mechanism, A and B were able to synchronize their copies of the data file using minimal network bandwidth.

Each node in Dynamo maintains a separate merkle tree for each key range it stores. Nodes compare whether keys in their respective key ranges are upto date using the tree traversal scheme explained above. It uses an intelligent partitioning scheme to avoid reconstruction of Merkle Trees every time the key ranges change (due to leaving and entering of nodes).

5. CONCLUSION

We have explored the breadths of the subject of Database Recovery Mechanisms by studying various recovery techniques used in traditional on-disk, Main Memory and Distributed database systems. In each of the preceding sections we have enumerated the different challenges faced by these systems in various settings. Solutions to these challenges make different assumptions about the underlying system model. We also observe how some of the modern systems compromise on strong guarantees for the sake of simplicity and performance. There seems to be a shift in thinking in the kind of solutions proposed in older versus more recent literature. Some of the seminal older research papers try to propose generalized monolithic solutions to the problem at hand, while the more recent ones try to solve more smaller and application specific problems.

6. REFERENCES

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *CIDR*, pages 9–20, January 2011.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[4] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro,

M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, June 1984.

[5] K. Elhardt and R. Bayer. A database cache for high performance and fast restart in database systems. *ACM Trans. Database Syst.*, 9(4):503–525, Dec. 1984.

[6] H. Garcia-Molina, R. J. Lipton, and J. Valdes. A massive memory machine. *IEEE Trans. Comput.*, 33(5):391–399, May 1984.

[7] L. H. Garcia-Munoz, J. E. Armendariz-Inigo, H. Decker, and F. D. Munoz-Escoi. Recovery protocols for replicated databases-a survey. In *Advanced Information Networking and Applications Workshops, 2007, AINAW '07. 21st International Conference on*, volume 1, pages 220–227, May 2007.

[8] J. Gray, P. Mcjones, M. Blasgen, B. Lindsay, R. Lorie, and T. Price. The recovery manager of the system r database manager. *ACM Computing Surveys*, 13:223–242, 1981.

[9] R. B. Hagmann. A crash recovery scheme for a memory-resident database system. *Computers, IEEE Transactions on*, 100(9):839–843, 1986.

[10] R. A. Lorie. Physical integrity in a large segmented database. *ACM Trans. Database Syst.*, 2(1):91–104, Mar. 1977.

[11] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.

[12] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[13] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* Osborne/McGraw-Hill, Berkeley, CA, USA, 2nd edition, 2000.

[14] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: A scalable log-structured database system in the cloud. *Proc. VLDB Endow.*, 5(10):1004–1015, June 2012.