Database Management Systems

Database Recovery

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
and
Centre for Artificial Intelligence and Machine Learning
Indian Statistical Institute, Kolkata

June, 2021

Basics

Why database recovery?

– To overcome the loss of information in case of any system crash or failure.

– To ensure that the atomicity and durability properties of transactions are preserved.

How?

By applying a recovery scheme.

What does a recovery scheme do?

– Restores the database to a consistent state that existed before the failure.

– Provides high availability (i.e. minimizes the time of unusability after a crash) of the database.

Types of failures

1. **Transaction failure:** A transaction might fail due to the following reasons –
   - Logical errors: Transaction cannot complete due to some internal error condition (e.g., bad input, data not found, overflow, shortage of resource limit, etc.)
   - System errors: An active transaction is terminated by the database system due to an error condition (e.g., deadlock).
2. **System crash:** A power failure or other hardware/software failure causes the loss of the volatile storage content.
3. **Disk failure:** A head crash or similar disk failure destroys all or part of disk storage. Destruction is assumed to be detectable and disk drives use checksums to detect failures.

Note: It is assumed that non-volatile storage contents will not corrupt due to system crash (known as *fail-stop assumption*) because integrity checks make a safeguard.

Working principle of recovery algorithms

Consider a transaction that transfers some PCs from the account ISI_{PC} to account $IISc_{PC}$.
– A failure may occur between one of these modifications have been made and both of them are made.
– Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
– Not modifying the database may result in lost updates if failure occurs just after transaction commits.

Recovery algorithms work in two ways:

1. Actions are taken during transaction processing to ensure that enough information exists to recover from failures (precaution)
2. Actions are taken post-failure to recover database contents to a state ensuring atomicity, consistency and durability (remedy)

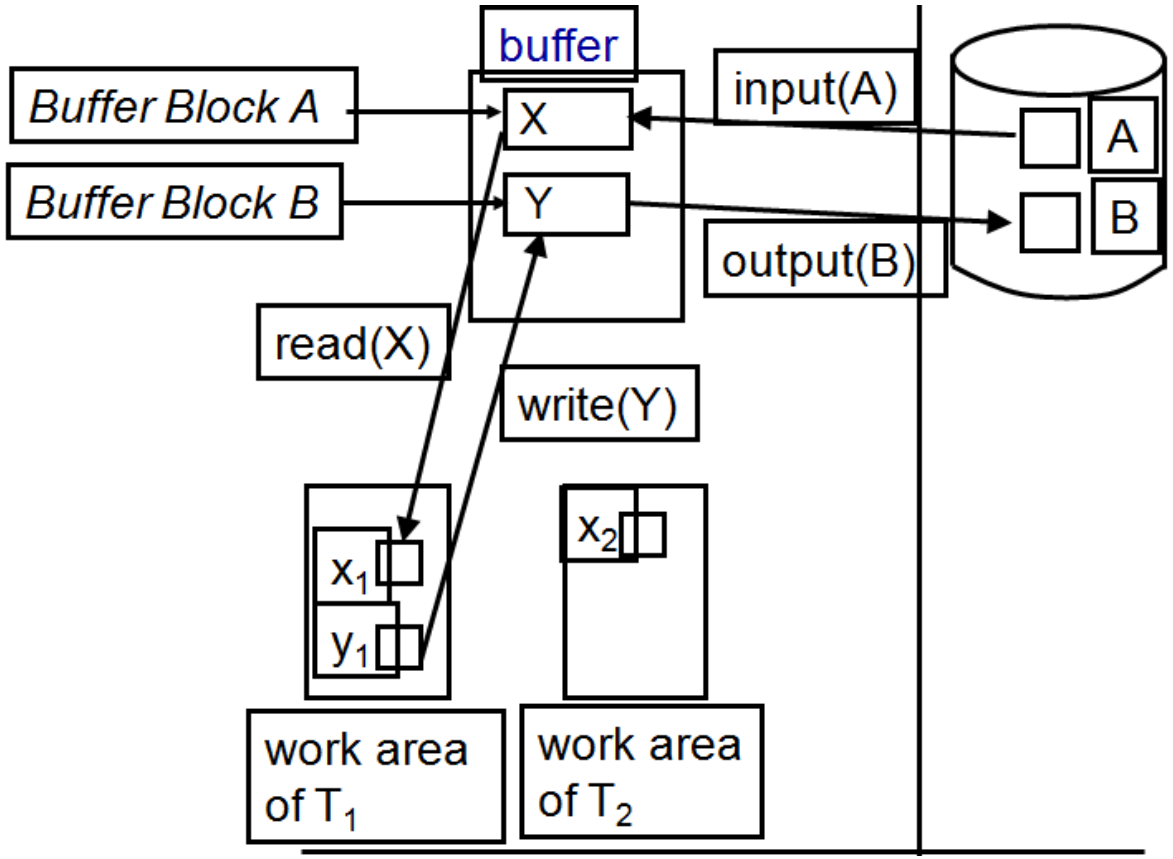
Types of storage

- **Volatile storage:** This type of storage does not survive from system crashes. E.g., main memory, cache memory, etc.
- **Nonvolatile storage:** These storages survive from system crashes, however, are subject to failure from head crash that may result in loss of information. E.g., disk, tape, flash memory, etc.
- **Stable storage:** This is a form of storage, although theoretically impossible to obtain, that survives from all failures by maintaining multiple copies on distinct nonvolatile media.

Implementation of stable storage

- Maintain multiple copies of each data block on separate disks (may be at remote sites)
- Partial/total failure during data transfer can still result in inconsistent copies. For this, we need to detect the data-transfer failure and accordingly restore the block to a consistent state. This is done as follows:
  – Keep multiple copies of logical database blocks (not like RAID where everything is local)
  – System maintains two physical blocks for each logical database block
  – For mirrored disks, both blocks are kept at the same location and for remote backup one block is local and the other one is remotely hosted
  – While operating, first write the information onto the first physical block, and when the first write successfully completes, write the same information onto the second physical block, and finally the output is considered as complete only after the second write successfully happens.
  – During recovery, the two physical blocks are compared and updated accordingly

Access to data block



Operations on block storage

Basics

Consider a transaction T_i that transfers 10 PCs from the account ISI_{PC} to account $IISc_{PC}$ having initial stocks of 50 and 200, respectively. Now, if the system crashes before $IISc_{PC}$ gets updated but after the write on ISI_{PC}, we could do the following for recovery:

1. **Execute nothing:** This will result into an inconsistent state by writing the values of ISI_{PC} and $IISc_{PC}$ as 40 and 200, respectively.
2. **Re-execute T_i:** This will result into an inconsistent state by writing the values of ISI_{PC} and $IISc_{PC}$ as 30 and 210, respectively.

Then, how to preserve the atomicity despite the failures?
We must first report the information describing the modifications to stable storage, instead of modifying the database itself.

Log-based recovery – Working principle

The log, which is kept on stable storage, is a sequence of *log records* that maintains a record of all types of update activities on the database to support efficient recovery.

The *log records* can be of the following types:

- Start log record: $<T_i$ start$>$, where T_i is the transaction identifier
- Update log reord: $<T_i, D_k, V, V'>$ or $<T_i, D_k, V'>$, where T_i, D_k, V, V' are the transaction identifier, data item identifier, old value of the data item and new value of the data item, respectively
- Abort log record: $<T_i$ abort$>$, where T_i is the transaction identifier
- Commit log record: $<T_i$ commit$>$, where T_i is the transaction identifier

Note: We assume that the transactions are executing serially.

Log-based recovery – Deferred database modification

This ensures transaction atomicity by recording all the database modifications in the log, but deferring the execution of all write operations pertaining to a transaction until it partially commits. In this scheme, the *update record log* appears as $<T_i, D_k, V'>$.

For recovery, we call redo(T_i) that sets the value of all data items to the new values as updated by transaction T_i. Transaction T_i needs to be redone iff the log contains both the *log records* $<T_i$ start$>$ and the record $<T_i$ commit$>$. This redo() should be idempotent, i.e. single execution must be logically equivalent to repeated executions.

Drawback: Overhead of storing local copies.

Log-based recovery – Deferred database modification

An example scenario: Assume $\text{ISI}_{PC} = 50$ and $\text{IISc}_{PC} = 200$.

Transactions (T_1, T_2)	**Entry to the log record**	**Write to database**	**Steps of recovery on failure**
read(ISI_{PC})	$<T_1$ start$>$		X(T_1)
$\text{ISI}_{PC} \leftarrow \text{ISI}_{PC}$ - 10	$<T_1, \text{ISI}_{PC}, 40>$		X(T_1)
write(ISI_{PC})			X(T_1)
read(IISc_{PC})			X(T_1)
$\text{IISc}_{PC} \leftarrow \text{IISc}_{PC} + 10$	$< T_1, \text{IISc}_{PC}, 210>$		X(T_1)
write(IISc_{PC})	$<T_1$ commit$>$		X(T_1)
		Yes	redo(T_1)
read(ISI_{PC})	$<T_2$ start$>$		redo(T_1), X(T_2)
$\text{ISI}_{PC} \leftarrow \text{ISI}_{PC}$ * 2	$<T_2, \text{ISI}_{PC}, 80>$		redo(T_1), X(T_2)
write(ISI_{PC})	$<T_2$ commit$>$		redo(T_1), X(T_2)
		Yes	redo(T_1), redo(T_2)

Note: X(T_i) denotes the removal of *log records* corresponding to the transaction T_i in execution.

Log-based recovery – Immediate database modification

This ensures transaction atomicity by updating an uncommitted transaction to the buffer, or to the disk itself before the transaction commits. In this scheme, the *update record log* appears as $< T_i, D_k, V, V' >$.

For recovery, we call either undo(T_i) that restores the value of all data items updated by transaction T_i to the old values, or redo(T_i) that sets the value of all data items to the new values as updated by transaction T_i. Transaction T_i needs to be undone if the log contains the *log record* $< T_i$ start$>$, but does not contain $< T_i$ commit$>$. Again, transaction T_i needs to be redone if the log contains both the *log records* $< T_i$ start$>$ and $< T_i$ commit$>$. Here also, the redo() and undo() should be idempotent.

Note: A transaction is said to have committed when its commit *log record* is sent to the stable storage.

Log-based recovery – Immediate database modification

An example scenario: Assume $\text{ISI}_{PC} = 50$ and $\text{IISc}_{PC} = 200$.

Transactions (T_1, T_2)	**Entry to the log record**	**Write to database**	**Steps of recovery on failure**
read(ISI_{PC})	$<T_1$ start$>$		undo(T_1)
$\text{ISI}_{PC} \leftarrow \text{ISI}_{PC}$ - 10	$<T_1, \text{ISI}_{PC}, 50, 40>$		undo(T_1)
write(ISI_{PC})		Yes	undo(T_1)
read(IISc_{PC})			undo(T_1)
$\text{IISc}_{PC} \leftarrow \text{IISc}_{PC} + 10$	$<T_1, \text{IISc}_{PC}, 200, 210>$		undo(T_1)
write(IISc_{PC})	$<T_1$ commit$>$		undo(T_1)
		Yes	redo(T_1)
read(ISI_{PC})	$<T_2$ start$>$		redo(T_1), undo(T_2)
$\text{ISI}_{PC} \leftarrow \text{ISI}_{PC}$ * 2	$<T_2, \text{ISI}_{PC}, 40, 80>$		redo(T_1), undo(T_2)
write(ISI_{PC})	$<T_2$ commit$>$		redo(T_1), undo(T_2)
		Yes	redo(T_1), redo(T_2)

Log-based recovery – Checkpoints

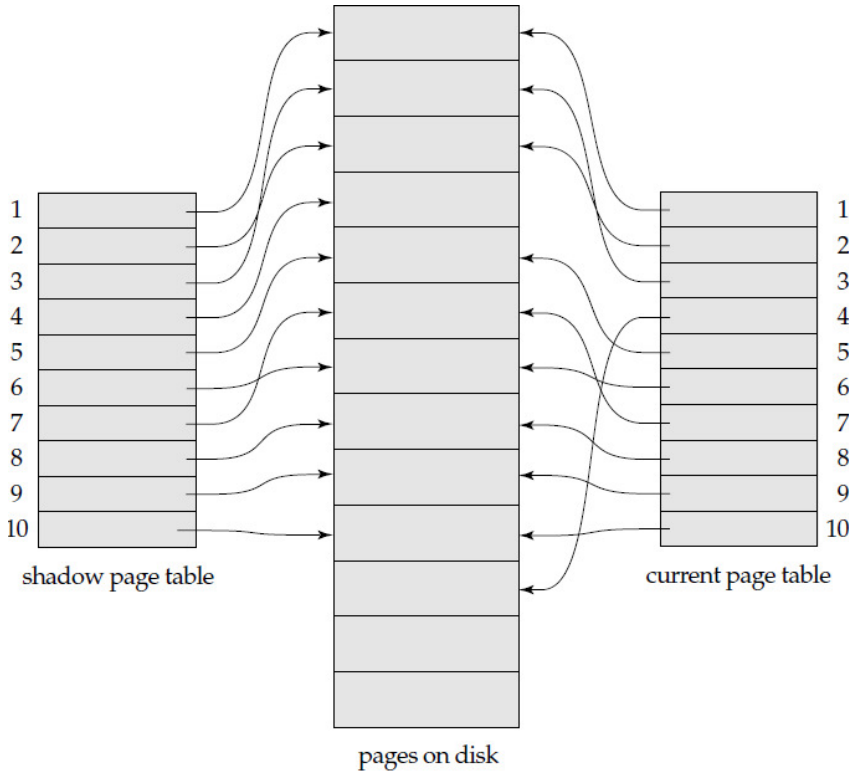
While recovering from the *log record*, we need to browse the entire log for undoing or redoing transactions.

This causes two problems:
– the search process becomes time consuming
– Some transactions that are required to be redone might have already written their updates to the database.

What is the solution then?
The system periodically performs checkpoints that involves – sending all *log records* from main memory to stable storage, output all modified buffer blocks to the disk and write a *log record* $<$ checkpoint $>$ to the stable storage.

Shadow paging – Working principle

Initialize identically and then maintain a shadow page table (never changed) and a current page table (changed when a transaction performs a write) during the transaction lifetime.



A view of the shadow and current page table

Basics

To take up recovery, in case of multiple concurrent transactions, we can use the log-based recovery scheme in a modified and extended form where –

1. the system should have a single disk buffer and a single log,
2. all transactions should share the buffer blocks, and
3. we have to allow immediate modification and permit a buffer block to have data items updated by one or more transactions.

Note: The assumption of stable storage also applies here.

Interaction with concurrency control

The recovery scheme should have a close interaction with the concurrency control.

We need to ensure that if a transaction T_i has updated a data item, no other transaction may update the same data item until T_i has committed or been rolled back. This means, updates on uncommitted transactions should not be visible to the other transactions.

What is the solution then?
We can use the strict two-phase locking protocol, i.e., two-phase locking with exclusive locks held until the end of the transaction.

Transaction rollback

A failed transaction T_i is rolled back by the recovery manager using *log records* as stated below.

1. Scan the log backward from the end
2. For each *log record* of the form $<T_i, D_k, V, V'>$ do the following:
   - Restore the data item D_k to its old value V
   - Write a *log record* $<T_i, D_k, V>$
   - On getting the *log record* $<T_i$ start$>$, stop the scan and write the *log record* $<T_i$ abort$>$

Note: Scanning the log backward is important, since a transaction may have updated a data item more than once.

Checkpoints

In a concurrent transaction-processing environment, several transactions might be active at the same time during the most recent checkpoint.

We can handle this by using the *log record* <checkpoint S>, where S is a set of transactions active at the time of the checkpoint. We assume that transactions do not perform updates either on the buffer blocks or on the log while the checkpoint is in progress.

Drawback: The requirement that transactions must not perform any updates to buffer blocks or to the log during checkpointing can be troublesome, since transaction processing will have to halt while a checkpoint is in progress.

Restart recovery – Primary steps

When the system recovers from a crash, it constructs two lists – The undo-list (consists of transactions to be undone) and the redo-list (consists of transactions to be redone).

Construction of the undo-list and redo-list:

1. Initialize both the undo-list and redo-list as empty.
2. Scan the log backward, examining each record, until it finds the first <checkpoint> record.
3. For each record found of the form $<T_i$ commit$>$, add T_i to the redo-list.
4. For each record found of the form $<T_i$ start$>$, if T_i is not in the redo-list then add T_i to the undo-list.
5. For each transaction T_i in the set S belonging to the checkpoint record, if T_i is not in the redo-list then add T_i to the undo-list.

Restart recovery – Steps for recovery

On preparing the redo-list and undo-list, the recovery scheme works as follows:

1. Rescan the log from the most recent record backward, and perform an undo for each log record that belongs transaction T_i on the undo-list. Log records of transactions on the redo-list are ignored in this phase. The scan stops when the $<T_i$ start$>$ records have been found for every transaction T_i in the undo-list.
2. Locate the most recent <checkpoint S> record on the log (this step may involve scanning the log forward, if the checkpoint record was passed in step 1).
3. Scan the log forward from the most recent <checkpoint S> record, and performs redo for each log record that belongs to a transaction T_i that is on the redo-list. It ignores log records of transactions on the undo-list in this phase.

Buffer management – Basics

Managing the buffer properly is required for the implementation of a crash-recovery scheme that ensures:

- data consistency
- imposes a minimal amount of overhead on interactions with the database.

A buffer manager is responsible for fetching data from disk storage into the main memory and deciding what data to cache in main memory. The involvement of buffer manager is important because it enables the database to handle data sizes that are much larger than the size of main memory.

Buffer management – Log-record buffering

Sending every *log record* to the stable storage might impose high overhead to the system.

What is the solution then?

Log records are buffered in main memory and are sent to the stable storage when a block of log records in the buffer is full, or a log force operation (commit a transaction by forcing all its *log records*) is executed.

Rules for log-record buffering:

1. *Log records* are output to stable storage in the order of their creation.
2. Transaction T_i enters the commit state only when the log record $<T_i$ commit$>$ has been output to stable storage.
3. Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage (known as write-ahead logging or WAL rule).

Buffer management – Database buffering

Database maintains an in-memory buffer of data blocks. When a new block is needed, an existing block is removed from the buffer if it is full. If the block chosen for removal has been updated, it must be output to the disk.

The rules for the output of *log records* limit the freedom of the system to output blocks of data. If the input of block B causes block B' to be chosen for output, all *log records* pertaining to data in B' must be output to the stable storage before B' is output. Thus, the sequence of actions by the system would be:

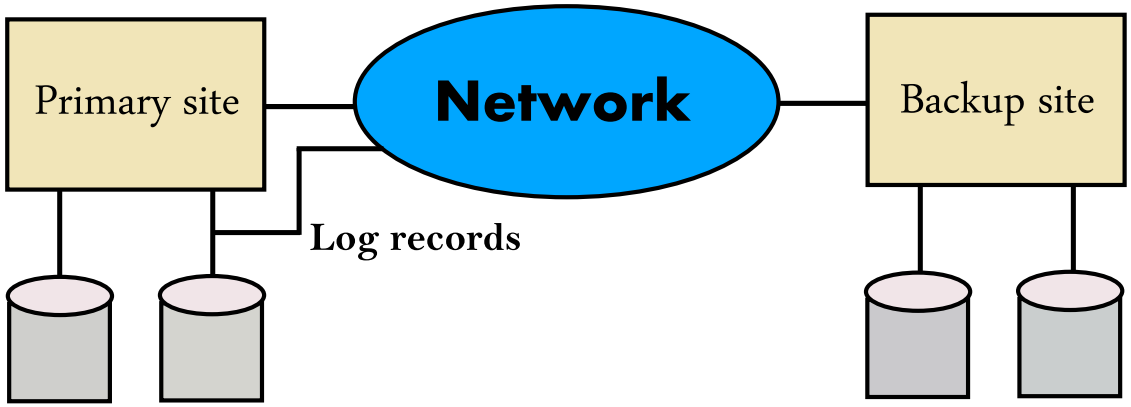
1. Output log records to stable storage until all *log records* pertaining to block B' have been output.
2. Output block B' to disk
3. Input block B from disk to main memory.

Buffer management – Role of OS

The buffer management can be managed by the operating system (OS) or by the database system itself.

- In the former case – The database system implements its buffer within the virtual memory provided by the OS. Since the OS knows about the entire memory requirements in the system, ideally it should be in charge of deciding what buffer blocks must be force-output to disk and when. But, to ensure the write-ahead logging requirements, the OS should request the database system to force-output the buffer blocks. The OS reserves swap space on disk for storing virtual memory pages that are not currently in main memory. This approach may result in extra output of data to disk (by the database besides the OS).
- In the latter case – The database system reserves part of main memory to serve as a buffer. This approach has the drawback of limiting flexibility in the use of main memory. The buffer must be kept small enough that other applications have sufficient main memory available for their needs.

Remote backup systems



Architecture of a remote backup system

When the primary site fails, the remote backup site takes over processing. In the beginning, it performs recovery based on the copy of the data and *log records* received from the primary site. Once recovery has been performed, the remote backup site starts processing transactions.

Advanced recovery techniques – Logical undo logging

To ensure recovery with concurrent transactions, we often use strict two-phase locking. But it can cause a significant decrease in concurrency when applied to certain specialized structures like B^+-tree index pages. To increase the concurrency, we can allow locks to be released early and use logical logging to manage undo operations efficiently.

In logical logging, if the operation inserted an entry in a B^+-tree, the undo information would indicate that a deletion operation is to be performed, and would identify the B^+-tree and what to delete from the tree.

Before any logical operation begins, it writes a log record $\langle T_i, O_j,$ operation-begin$\rangle$, where O_j is the unique identifier for the operation.

Advanced recovery techniques – Fuzzy checkpointing

Checkpointing in concurrent environment requires that all updates to the database be temporarily suspended while the checkpoint is in progress. For large number of pages in the buffer, a checkpoint may take a long time to finish. Fuzzy checkpoints are used to relax this.

Fuzzy checkpointing allows updates to start once the checkpoint record has been written, but before the modified buffer blocks are written to disk.

Here, the location of the last completed checkpoint is stored in a fixed position (known as last-checkpoint) on disk. The system does not update this information when it writes the checkpoint record. Instead, it creates a list of all modified buffer blocks and update the last-checkpoint information only after all buffer blocks in the list of modified buffer blocks have been output to disk.

Advanced recovery techniques – ARIES

This is a complex recovery scheme that is widely used in industry. It uses a log sequence number (LSN) to identify *log records* and a dirty page table to minimize unnecessary redos during recovery, employs fuzzy checkpointing, and supports physiological redo operations. It also incorporates a number of optimizations to minimize the recovery time.

Key features of ARIES:

- Physical and operation logging
- Page oriented redo and logical undo
- WAL and in-place updates
- Transaction rollback
- Fine-grain concurrency control