Chapter 19
Database Recovery Techniques

Database Recovery

Purpose of Database Recovery
- To bring the database into the last consistent state, which existed prior to the failure.
- To preserve transaction properties (Atomicity, Consistency, Isolation and Durability).
- Example:
  - If the system crashes before a fund transfer transaction completes its execution, then either one or both accounts may have an incorrect value. Thus, the database must be restored to the state before the transaction modified any of the accounts.

Types of Failure
- The database may become unavailable for use due to
  - Transaction failure: Transactions may fail because of incorrect input, deadlock, incorrect synchronization.
  - System failure: System may fail because of addressing error, application error, operating system fault, RAM failure, etc.
  - Media failure: Disk head crash, power disruption, etc.
Database Recovery

Transaction Log
- For recovery from any type of failure data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFler Image) are required.
- These values and other information is stored in a sequential file called Transaction log.

Database Recovery

Data Caching
- Data items to be modified are first stored into database cache by the Cache Manager (CM) and after modification they are flushed (written) to the disk.
- The flushing is controlled by Modified and Pin-Unpin bits.
  - Pin-Unpin: Instructs the operating system not to flush the data item.
  - Modified: Indicates the modification of the data item.

Database Recovery

Data Update
- Immediate Update: As soon as a data item is modified in cache, the disk copy is updated.
- Deferred Update: All modified data items in the cache are written either after a transaction ends its execution or after a fixed number of transactions have completed their execution.
- Shadow update: The modified version of a data item does not overwrite its disk copy but is written at a separate disk location.
- In-place update: The disk version of the data item is overwritten by the cache version.
### Database Recovery

**Transaction Roll-back (Undo) and Roll-Forward (Redo)**

- To maintain atomicity, a transaction’s operations are redone or undone.
  - **Undo**: Restore all BFIMs on to disk (Remove all AFIMs).
  - **Redo**: Restore all AFIMs on to disk.

- Database recovery is achieved either by performing only Undos or only Redos or by a combination of the two. These operations are recorded in the log as they happen.

### Database Recovery

**Write-Ahead Logging**

- When in-place update (immediate or deferred) is used then log is necessary for recovery and it must be available to recovery manager. This is achieved by **Write-Ahead Logging (WAL)** protocol. WAL states that
  - For **Undo**: Before a data item’s AFIM is flushed to the database disk (overwriting the BFIM) its BFIM must be written to the log and the log must be saved on a stable store (log disk).
  - For **Redo**: Before a transaction executes its commit operation, all its AFIMs must be written to the log and the log must be saved on a stable store.

### Database Recovery

**Checkpointing**

- Time to time (randomly or under some criteria) the database flushes its buffer to database disk to minimize the task of recovery. The following steps defines a checkpoint operation:
  1. Suspend execution of transactions temporarily.
  2. Force write modified buffer data to disk.
  3. Write a [checkpoint] record to the log, save the log to disk.
  4. Resume normal transaction execution.

- During recovery redo or undo is required to transactions appearing after [checkpoint] record.
Database Recovery

Recovery Scheme

- Deferred Update (No Undo/Redo)
  - The data update goes as follows:
    - A set of transactions records their updates in the log.
    - At commit point under WAL scheme these updates are saved on database disk.
    - After reboot from a failure the log is used to redo all the transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

- Deferred Update in a single-user system
  - There is no concurrent data sharing in a single user system. The data update goes as follows:
    - A set of transactions records their updates in the log.
    - At commit point under WAL scheme these updates are saved on database disk.
    - After reboot from a failure the log is used to redo all the transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

- Deferred Update with concurrent users
  - This environment requires some concurrency control mechanism to guarantee isolation property of transactions. In a system recovery, transactions which were recorded in the log after the last checkpoint were redone. The recovery manager may scan some of the transactions recorded before the checkpoint to get the AFIMs.
### Database Recovery

#### Deferred Update with concurrent users
- Two tables are required for implementing this protocol:
  - **Active table**: All active transactions are entered in this table.
  - **Commit table**: Transactions to be committed are entered in this table.
- During recovery, all transactions of the commit table are redone and all transactions of active tables are ignored since none of their AFIMs reached the database. It is possible that a commit table transaction may be redone twice but this does not create any inconsistency because a redone is "idempotent", that is, one redone for an AFIM is equivalent to multiple redone for the same AFIM.

### Database Recovery

#### Recovery Techniques Based on Immediate Update
- **Undo/No-redo Algorithm**
  - In this algorithm AFIMs of a transaction are flushed to the database disk under WAL before it commits.
  - For this reason the recovery manager undoes all transactions during recovery.
  - No transaction is redone.
  - It is possible that a transaction might have completed execution and ready to commit but this transaction is also undone.

### Database Recovery

#### Recovery Techniques Based on Immediate Update
- **Undo/Redo Algorithm** (Single-user environment)
  - Recovery schemes of this category apply undo and also redo for recovery.
  - In a single-user environment no concurrency control is required but a log is maintained under WAL.
  - Note that at any time there will be one transaction in the system and it will be either in the commit table or in the active table.
  - The recovery manager performs:
    - Undo of a transaction if it is in the active table.
    - Redo of a transaction if it is in the commit table.
Database Recovery

Recovery Techniques Based on Immediate Update

- **Undo/Redo Algorithm (Concurrent execution)**
- Recovery schemes of this category apply undo and also redo to recover the database from failure.
- In concurrent execution environment a concurrency control is required and log is maintained under WAL.
- Commit table records transactions to be committed and active table records active transactions. To minimize the work of the recovery manager, checkpointing is used.
- The recovery performs:
  - **Undo** of a transaction if it is in the active table.
  - **Redo** of a transaction if it is in the commit table.

Database Recovery

Shadow Paging

- The AFIM does not overwrite its BFIM but recorded at another place on the disk. Thus, at any time a data item has AFIM and BFIM (Shadow copy of the data item) at two different places on the disk.

![Database Diagram](image)

**X and Y**: Shadow copies of data items

**X' and Y'**: Current copies of data items

Database Recovery

Shadow Paging

- To manage access of data items by concurrent transactions two directories (current and shadow) are used.
  - The directory arrangement is illustrated below. Here a page is a data item.
Database Recovery

Recovery in multidatabase system
- A multidatabase system is a special distributed database system where one node may be running relational database system under UNIX, another may be running object-oriented system under Windows and so on.
- A transaction may run in a distributed fashion at multiple nodes.
- In this execution scenario the transaction commits only when all these multiple nodes agree to commit individually the part of the transaction they were executing.
- This commit scheme is referred to as “two-phase commit” (2PC).
  - If any one of these nodes fails or cannot commit the part of the transaction, then the transaction is aborted.
- Each node recovers the transaction under its own recovery protocol.

Atomicity

- Ensuring atomicity in a distributed system requires a transaction coordinator, which is responsible for the following:
  - Starting the execution of the transaction
  - Breaking the transaction into a number of subtransactions, and distribution of these subtransactions to the appropriate sites for execution
  - Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites

Two-Phase Commit Protocol (2PC)

- Assumes fail-stop model
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached
- When the protocol is initiated, the transaction may still be executing at some of the local sites
- The protocol involves all the local sites at which the transaction executed
- Example: Let T be a transaction initiated at site S_i and let the transaction coordinator at S_j be C_j.
Phase 1: Obtaining a Decision

- $C_i$ adds <prepare $T$> record to the log
- $C_i$ sends <prepare $T$> message to all sites
- When a site receives a <prepare $T$> message, the transaction manager determines if it can commit the transaction
  - If no: add <no $T$> record to the log and respond to $C_i$ with <abort $T$>
  - If yes:
    - add <ready $T$> record to the log
    - force all log records for $T$ onto stable storage
    - send <ready $T$> message to $C_i$

Phase 1 (Cont.)

- Coordinator collects responses
  - All respond "ready", decision is commit
  - At least one response is "abort", decision is abort
  - At least one participant fails to respond within time out period, decision is abort

Phase 2: Recording Decision in the Database

- Coordinator adds a decision record <abort $T$> or <commit $T$> to its log and forces record onto stable storage
- Once that record reaches stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally
Failure Handling in 2PC – Site Failure

- The log contains a \(<\text{commit}\ T>\) record
  - In this case, the site executes \(\text{redo}(T)\)
- The log contains an \(<\text{abort}\ T>\) record
  - In this case, the site executes \(\text{undo}(T)\)
- The log contains a \(<\text{ready}\ T>\) record; consult \(C_i\)
  - If \(C_i\) is down, site sends \text{query-status} T message to the other sites
- The log contains no control records concerning \(T\)
  - In this case, the site executes \(\text{undo}(T)\)

Failure Handling in 2PC – Coordinator \(C_i\) Failure

- If an active site contains a \(<\text{commit}\ T>\) record in its log, then \(T\) must be committed
- If an active site contains an \(<\text{abort}\ T>\) record in its log, then \(T\) must be aborted
- If some active site does not contain the record \(<\text{ready}\ T>\) in its log then the failed coordinator \(C_i\) cannot have decided to commit \(T\)
  - Rather than wait for \(C_i\) to recover, it is preferable to abort \(T\)
- All active sites have a \(<\text{ready}\ T>\) record in their logs, but no additional control records
  - In this case we must wait for the coordinator to recover
  - Blocking problem – \(T\) is blocked pending the recovery of site \(S_i\)