Outline

- Introduction
- Background
- Distributed DBMS Architecture
- Distributed Database Design
- Distributed Query Processing
- Distributed Transaction Management
  - Transaction Concepts and Models
  - Distributed Concurrency Control
  - Distributed Reliability
- Building Distributed Database Systems (RAID)
- Mobile Database Systems
- Privacy, Trust, and Authentication
- Peer to Peer Systems

Useful References

- Textbook *Principles of Distributed Database Systems*,

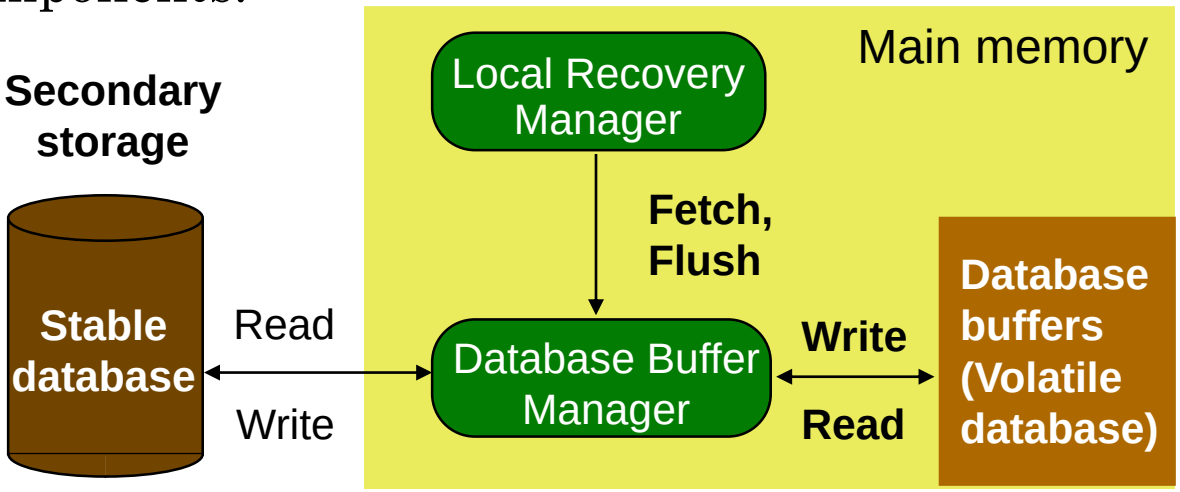
  Chapter 12.3

Types of Failures

- Transaction failures
  - Transaction aborts (unilaterally or due to deadlock)
  - Avg. 3% of transactions abort abnormally
- System (site) failures
  - Failure of processor, main memory, power supply, …
  - Main memory contents are lost, but secondary storage contents are safe
  - Partial vs. total failure
- Media failures
  - Failure of secondary storage devices such that the stored data is lost
  - Head crash/controller failure (?)
- Communication failures
  - Lost/undeliverable messages
  - Network partitioning

Local Recovery Management – Architecture

- Volatile storage
  - Consists of the main memory of the computer system (RAM).
- Stable storage
  - Resilient to failures and loses its contents only in the presence of media failures (e.g., head crashes on disks).
  - Implemented via a combination of hardware (non-volatile storage) and software (stable-write, stable-read, clean-up) components.



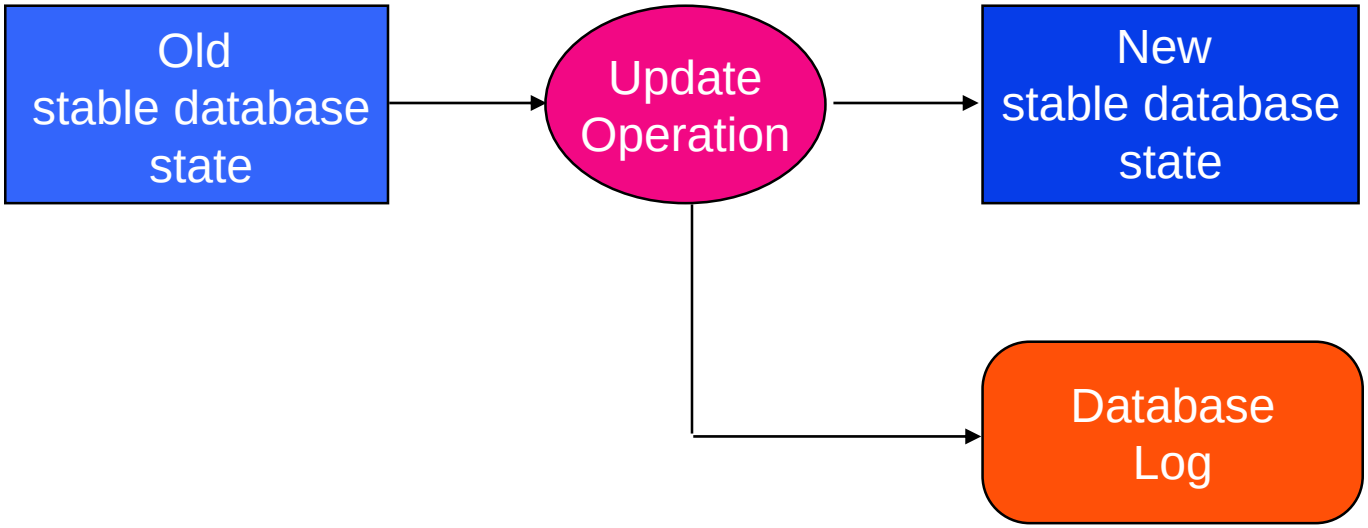
Update Strategies

- In-place update
  - Each update causes a change in one or more data values on pages in the database buffers
- Out-of-place update
  - Each update causes the new value(s) of data item(s) to be stored separate from the old value(s)

In-Place Update Recovery Information

Database Log

Every action of a transaction must not only perform the action, but must also write a *log* record to an append-only file.



Logging

The log contains information used by the recovery process to restore the consistency of a system. This information may include

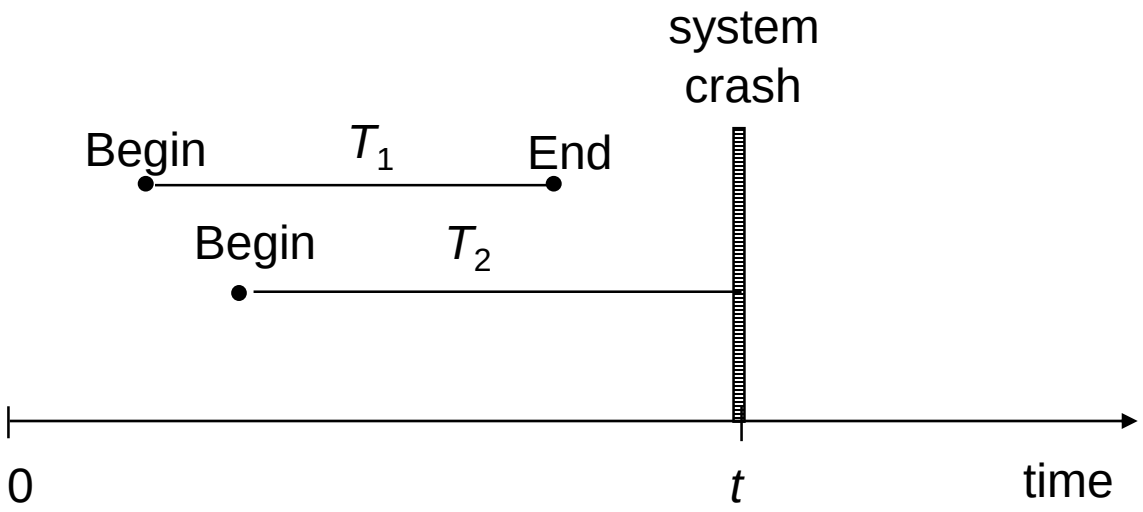
- transaction identifier
- type of operation (action)
- items accessed by the transaction to perform the action
- old value (state) of item (before image)
- new value (state) of item (after image)

...

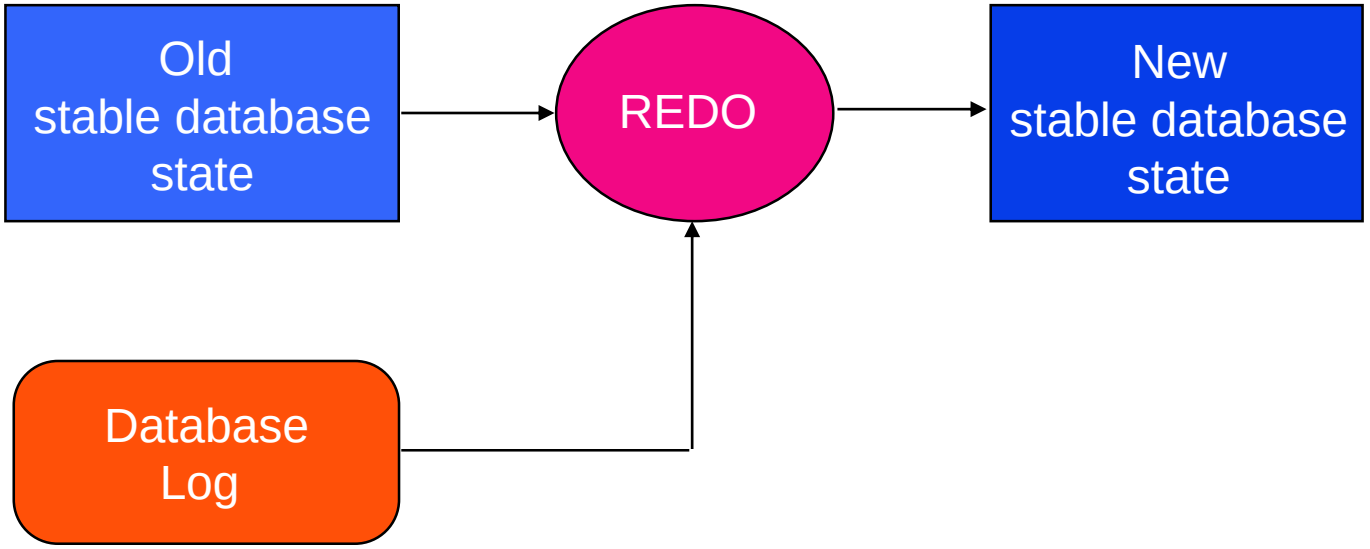
Why Logging?

Upon recovery:

- all of T_1's effects should be reflected in the database (REDO if necessary due to a failure)
- none of T_2's effects should be reflected in the database (UNDO if necessary)

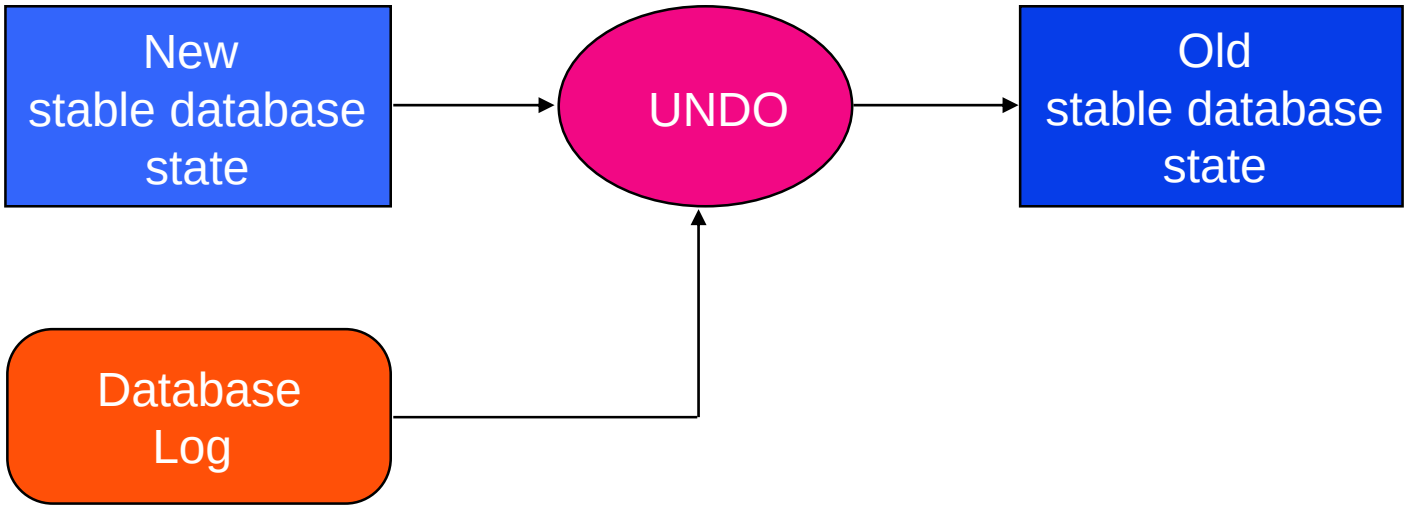


REDO Protocol



- REDO'ing an action means performing it again.
- The REDO operation uses the log information and performs the action that might have been done before, or not done due to failures.
- The REDO operation generates the new image.

UNDO Protocol



- UNDO'ing an action means to restore the object to its before image.
- The UNDO operation uses the log information and restores the old value of the object.

When to Write Log Records Into Stable Store

Assume a transaction T updates a page P

- Fortunate case
  - System writes P in stable database
  - System updates stable log for this update
  - SYSTEM FAILURE OCCURS!... (before T commits)

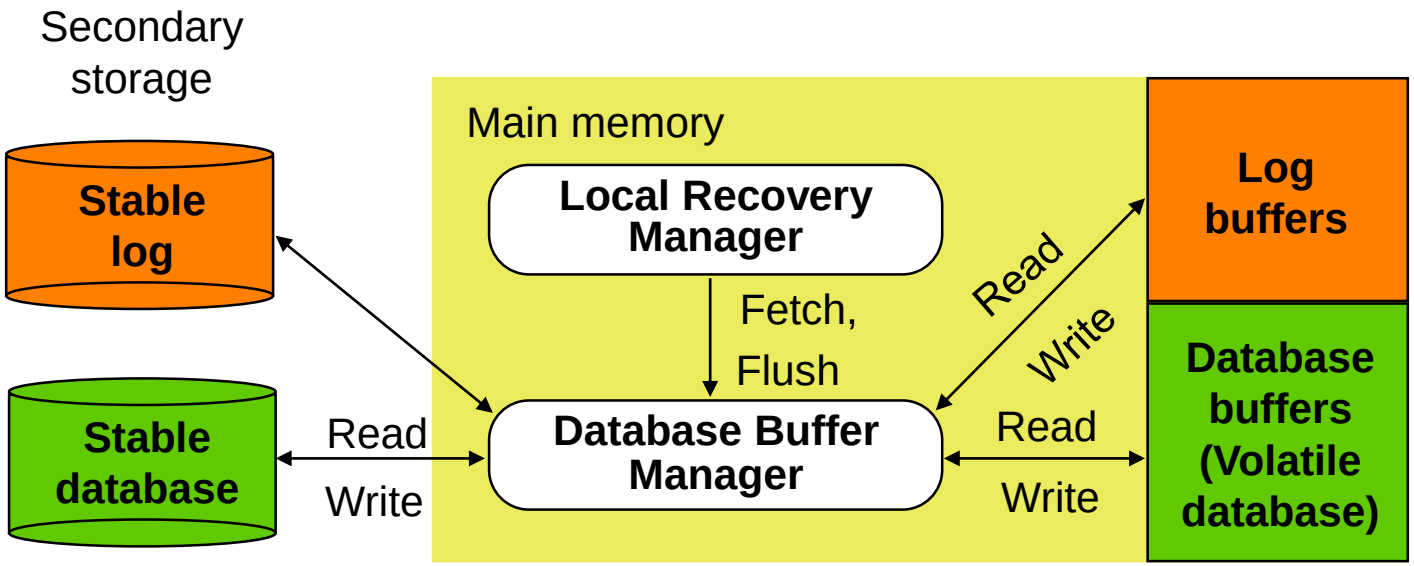
  We can recover (undo) by restoring P to its old state by using the log
- Unfortunate case
  - System writes P in stable database
  - SYSTEM FAILURE OCCURS!... (before stable log is updated)

  We cannot recover from this failure because there is no log record to restore the old value.
- Solution: Write-Ahead Log (WAL) protocol

Write–Ahead Log Protocol

- Notice:
  - If a system crashes before a transaction is committed, then all the operations must be undone. Only need the before images (*undo portion* of the log).
  - Once a transaction is committed, some of its actions might have to be redone. Need the after images (*redo portion* of the log).
- WAL protocol :
  - Before a stable database is updated, the undo portion of the log should be written to the stable log
  - When a transaction commits, the redo portion of the log must be written to stable log prior to the updating of the stable database.

Logging Interface (see book)



Out-of-Place Update Recovery Information (see book)

- Shadowing
  - When an update occurs, don't change the old page, but create a shadow page with the new values and write it into the stable database.
  - Update the access paths so that subsequent accesses are to the new shadow page.
  - The old page retained for recovery.
- Differential files
  - For each file F maintain
    - a read only part FR
    - a differential file consisting of insertions part DF+ and deletions part DF-
    - Thus, $F = (FR \cup DF+) - DF-$
  - Updates treated as delete old value, insert new value

Execution of Commands (see book)

Commands to consider:

- begin_transaction
- read
- write
- commit
- abort
- recover

Independent of execution strategy for LRM

Execution Strategies (see book)

- Dependent upon
  - Can the buffer manager decide to write some of the buffer pages being accessed by a transaction into stable storage or does it wait for LRM to instruct it?
    - fix/no-fix decision
  - Does the LRM force the buffer manager to write certain buffer pages into stable database at the end of a transaction's execution?
    - flush/no-flush decision
- Possible execution strategies:
  - no-fix/no-flush
  - no-fix/flush
  - fix/no-flush
  - fix/flush

No-Fix/No-Flush (see book)

- Abort
  - Buffer manager may have written some of the updated pages into stable database
  - LRM performs transaction undo (or partial undo)
- Commit
  - LRM writes an "end_of_transaction" record into the log.
- Recover
  - For those transactions that have both a "begin_transaction" and an "end_of_transaction" record in the log, a partial redo is initiated by LRM
  - For those transactions that only have a "begin_transaction" in the log, a global undo is executed by LRM

No-Fix/Flush (see book)

- Abort
  - Buffer manager may have written some of the updated pages into stable database
  - LRM performs transaction undo (or partial undo)
- Commit
  - LRM issues a `flush` command to the buffer manager for all updated pages
  - LRM writes an "end_of_transaction" record into the log.
- Recover
  - No need to perform redo
  - Perform global undo

Fix/No-Flush (see book)

- Abort
  - None of the updated pages have been written into stable database
  - Release the `fix`ed pages
- Commit
  - LRM writes an "end_of_transaction" record into the log.
  - LRM sends an `unfix` command to the buffer manager for all pages that were previously `fix`ed
- Recover
  - Perform partial redo
  - No need to perform global undo

Fix/Flush (see book)

- Abort
  - None of the updated pages have been written into stable database
  - Release the `fix`ed pages
- Commit (the following have to be done atomically)
  - LRM issues a `flush` command to the buffer manager for all updated pages
  - LRM sends an `unfix` command to the buffer manager for all pages that were previously `fix`ed
  - LRM writes an “end_of_transaction” record into the log.
- Recover
  - No need to do anything

Checkpoints

- Simplifies the task of determining actions of transactions that need to be undone or redone when a failure occurs.
- A checkpoint record contains a list of active transactions.
- Steps:
  - Write a begin_checkpoint record into the log
  - Collect the checkpoint dat into the stable storage
  - Write an end_checkpoint record into the log

Media Failures –Full Architecture (see book)