ST. XAVIER’S COLLEGE

**Maitighar, Kathmandu**

**(Affiliated to Tribhuvan University)**



**Database Management System**

**Theory Assignment #9**

**Submitted By**

Alok Shrestha

013BSCIT005

B.Sc. CSIT Year II/IV Semester

**Submitted To**

Er. Sanjay Kumar Yadav

Lecturer

Department of Computer Science

St. Xavier’s College

Maitighar, Kathmandu

**Submitted On**

October 2015

**Recovery System**

**Types of Failure**

1. **Transaction failure** :

* **Logical errors**: transaction cannot complete due to some internal error condition
* **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)

1. **System crash**: a power failure or other hardware or software failure causes the system to crash.

* **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash

Database systems have numerous integrity checks to prevent corruption of disk data

1. **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage

* Destruction is assumed to be detectable: disk drives use checksums to detect failures

# Recovery Algorithms

* Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
* Recovery algorithms have two parts

1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

# Storage Structure

* **Volatile storage**:
  + does not survive system crashes
  + examples: main memory, cache memory
* **Nonvolatile storage**:
  + survives system crashes
  + Examples: disk, tape, flash memory,
  + non-volatile (battery backed up) RAM
* **Stable storage**:
  + a mythical form of storage that survives all failures
  + approximated by maintaining multiple copies on distinct nonvolatile media

# Stable-Storage Implementation

* Maintain multiple copies of each block on separate disks z copies can be at remote sites to protect against disasters such as fire or flooding.
* Failure during data transfer can still result in inconsistent copies: Block transfer can result in
* Successful completion
* Partial failure: destination block has incorrect information
* Total failure: destination block was never updated
* Protecting storage media from failure during data transfer (one solution):
  + - Execute output operation as follows (assuming two copies of each block):

1. Write the information onto the first physical block.
2. When the first write successfully completes, write the same information onto the second physical block.
3. The output is completed only after the second write successfully completes.

* Protecting storage media from failure during data transfer (cont.):
* Copies of a block may differ due to failure during output operation. To recover from failure:

1. First find inconsistent blocks:

*Expensive solution*: Compare the two copies of every disk block.

*Better solution*:

* Record in-progress disk writes on non-volatile storage (Nonvolatile RAM or special area of disk).
* Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
* Used in hardware RAID systems

1. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

# Data Access

* **Physical blocks** are those blocks residing on the disk.
* **Buffer blocks** are the blocks residing temporarily in main memory.
* Block movements between disk and main memory are initiated through the following two operations:
* **input**(*B*) transfers the physical block *B* to main memory.
* **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.

# Recovery and Atomicity

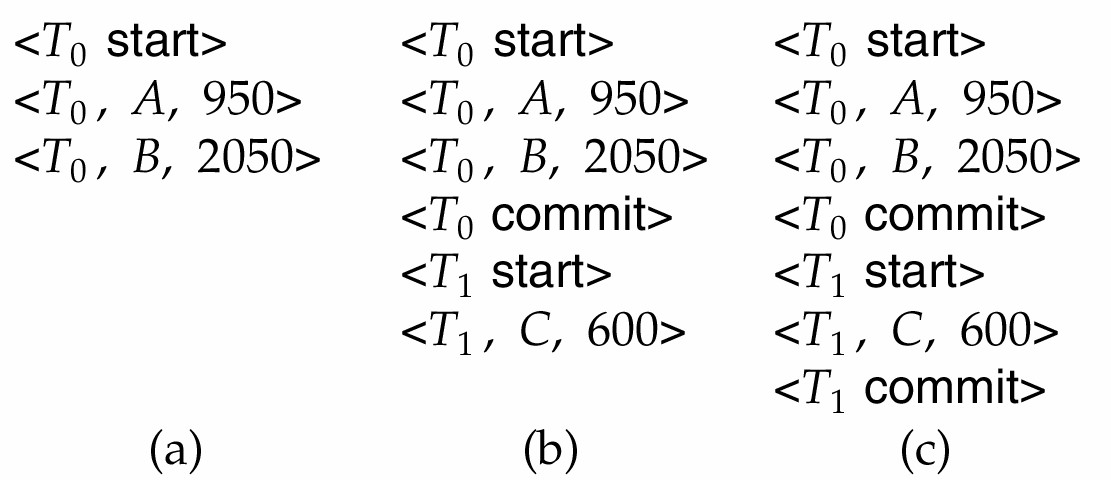
* Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
* Consider transaction *Ti* that transfers $50 from account *A* to account *B*; goal is either to perform all database modifications made by *Ti* or none at all.
* Several output operations may be required for *Ti* (to output *A* and *B*). A failure may occur after one of these modifications has been made but before all of them are made.
* To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
* We study two approaches: **log-based recovery**, and **shadow-paging**
* We assume (initially) that transactions run serially, that is, one after the other.

# Log-Based Recovery

* A logis kept on stable storage.
* The log is a sequence of **log records**, and maintains a record of update activities on the database.
* When transaction *Ti* starts, it registers itself by writing a *<Ti* **start**>log record
* *Before Ti* executes **write**(*X*), a log record *<Ti, X, V1, V2>*is written, where *V1* is the value of *X* before the write, and *V2* is the value to be written to *X*.
* Log record notes that *Ti* has performed a write on data item *Xj Xj* had value *V1* before the write, and will have value *V2* after the write.
* When *Ti* finishes it last statement, the log record <*Ti* **commi**t> is written.
* We assume for now that log records are written directly to stable storage (that is, they are not buffered)
* Two approaches using logs
  + Deferred database modification
  + Immediate database modification

# Deferred Database Modification

* The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
* Assume that transactions execute serially
* Transaction starts by writing *<Ti* ***start****>*record to log.
* A **write**(*X*) operation results in a log record *<Ti, X, V>*being written, where *V* is the new value for *X* z Note: old value is not needed for this scheme
* The write is not performed on *X* at this time, but is deferred.
* When *Ti* partially commits, <*Ti* **commit**> is written to the log
* Finally, the log records are read and used to actually execute the previously deferred writes.
* During recovery after a crash, a transaction needs to be redone if and only if both *<Ti* **start**> and<*Ti* **commit**> are there in the log.
* Redoing a transaction *Ti* ( **redo***Ti*) sets the value of all data items updated by the transaction to the new values.
* Crashes can occur while z the transaction is executing the original updates, or z while recovery action is being taken
* example transactions *T0* and *T1* (*T0* executes before *T1*):
  + *T0*: **read** (*A*) *T1* : **read** (*C*)
* *A: - A - 50 C:- C- 100* **Write** (*A*) **write** (*C*) **read** (*B*) *B:- B + 50* **write** (*B*)
* Below we show the log as it appears at three instances of time.



* If log on stable storage at time of crash is as in case:

1. No redo actions need to be taken
2. redo(*T*0) must be performed since <*T*0 **commi**t> is present
3. **redo**(*T*0) must be performed followed by redo(*T*1) since <*T*0 **commit**> and <*Ti* commit> are present

# Immediate Database Modification

* The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
* since undoing may be needed, update logs must have both old and new value
* Update log record must be written *before* database item is written
* We assume that the log record is output directly to stable storage
* Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
* Output of updated blocks can take place at any time before or after transaction commit
* Order in which blocks are output can be different from the order in which they are written.

**Immediate Database Modification Example**

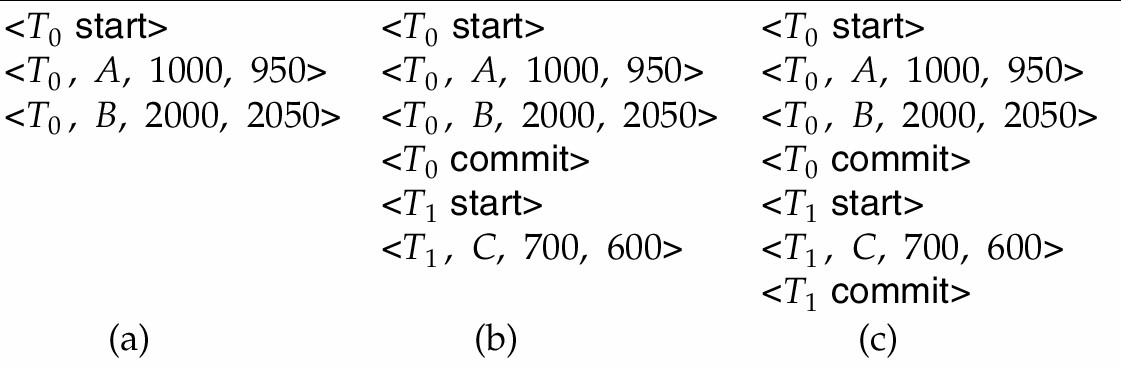
# 

* Recovery procedure has two operations instead of one:
  + **undo**(*T*i) restores the value of all data items updated by *Ti* to their old values, going backwards from the last log record for *Ti*
  + **redo**(*T*i) sets the value of all data items updated by *Ti* to the new values, going forward from the first log record for *Ti*
* Both operations must be **idempotent**
* That is, even if the operation is executed multiple times the effect is the same as if it is executed once
* Needed since operations may get re-executed during recovery „When recovering after failure:
  + Transaction *Ti* needs to be undone if the log contains the record *<Ti* **start***>*, but does not contain the record *<Ti* **commit***>*.
  + Transaction *Ti* needs to be redone if the log contains both the record *<Ti* **start***>*and the record *<Ti* **commit***>*.
* Undo operations are performed first, then redo operations.

## **Immediate DB Modification Recovery**

**Example**

Below we show the log as it appears at three instances of time.



Recovery actions in each case above are:

1. undo (*T*0): B is restored to 2000 and A to 1000.
2. undo (*T*1) and redo (*T*0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.
3. redo (*T*0) and redo (*T*1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

# Checkpoints

* Problems in recovery procedure as discussed earlier :

1. searching the entire log is time-consuming
2. we might unnecessarily redo transactions which have already
3. output their updates to the database.

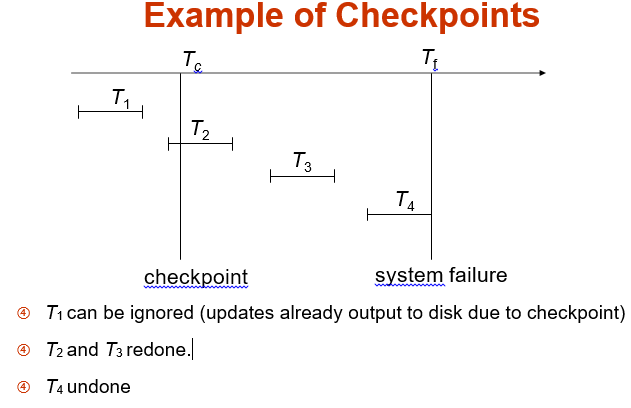
* Streamline recovery procedure by periodically performing **checkpointing**

1. Output all log records currently residing in main memory onto stable storage.
2. Output all modified buffer blocks to the disk.
3. Write a log record <**checkpoint**> onto stable storage.

During recovery we need to consider only the most recent transaction Ti that started before the checkpoint, and transactions that started after *Ti*.

1. Scan backwards from end of log to find the most recent <**checkpoint**> record
2. Continue scanning backwards till a record *<Ti* **start**> is found.
3. Need only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
4. For all transactions (starting from *Ti* or later) with no *<Ti* **commit***>*, execute **undo*(****Ti).* (Done only in case of immediate modification.)
5. Scanning forward in the log, for all transactions starting from *Ti* or later with a *<Ti* commit*>*, execute redo*(Ti).*

**Example of Checkpoints**



# Log Record Buffering

* **Log record buffering**: log records are buffered in main memory, instead of of being output directly to stable storage.
  + Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
* Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
* Several log records can thus be output using a single output operation, reducing the I/O cost.

The rules below must be followed if log records are buffered:

* Log records are output to stable storage in the order in which they are created.
* Transaction *Ti* enters the commit state only when the log record <*Ti* **commit**> has been output to stable storage.
* 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.

This rule is called the **write-ahead logging** or **WAL** rule – Strictly speaking WAL only requires undo information to be output

# Database Buffering

* Database maintains an in-memory buffer of data blocks z When a new block is needed, if buffer is full an existing block needs to be removed from buffer
* If the block chosen for removal has been updated, it must be output to disk
* If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first(Write ahead logging)
* No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
* Before writing a data item, transaction acquires exclusive lock on block containing the data item. Lock can be released once the write is completed.
* Such locks held for short duration are called **latches**.
* Before a block is output to disk, the system acquires an exclusive latch on the block
* Ensures no update can be in progress on the block
* Database buffer can be implemented either
  + in an area of real main-memory reserved for the database, or
  + in virtual memory
* Implementing buffer in reserved main-memory has drawbacks:
  + Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  + Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.
* Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  + When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  + When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
  + Known as **dual paging** problem.
  + Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    - Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    - Release the page from the buffer, for the OS to use
  + Dual paging can thus be avoided, but common operating systems do not support such functionality.

# Failure with Loss of Nonvolatile Storage

* So far we assumed no loss of non-volatile storage
* Technique similar to checkpointing used to deal with loss of nonvolatile storage
  + Periodically **dump** the entire content of the database to stable storage
  + No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
* Output all log records currently residing in main memory onto stable storage.
* Output all buffer blocks onto the disk.
* Copy the contents of the database to stable storage.
* Output a record <**dump**> to log on stable storage.

## Recovering from Failure of Non-Volatile Storage

* To recover from disk failure
  + restore database from most recent dump.
  + Consult the log and redo all transactions that committed after the dump
* Can be extended to allow transactions to be active during dump; known as **fuzzy dump** or **online dump**
* Support for high-concurrency locking techniques, such as those used for B+-tree concurrency control, which release locks early
  + Supports “logical undo”
* Recovery based on “repeating history”, whereby recovery executes exactly the same actions as normal processing
  + including redo of log records of incomplete transactions, followed by subsequent undo
  + Key benefits
    - supports logical undo
    - easier to understand/show correctness

## **Advanced Recovery: Logical Undo Logging**

* Operations like B+-tree insertions and deletions release locks early.
  + They cannot be undone by restoring old values (**physical undo**), since once a lock is released, other transactions may have updated the B+-tree.
  + Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as **logical undo**).
* For such operations, undo log records should contain the undo operation to be executed
  + Such logging is called **logical undo logging**, in contrast to **physical undo logging**
* Operations are called **logical operations**
* Other examples:
  + delete of tuple, to undo insert of tuple
  + allows early lock release on space allocation information
  + subtract amount deposited, to undo deposit
  + allows early lock release on bank balance

# Advanced Recovery: Checkpointing

* **Checkpointing** is done as follows:

1. Output all log records in memory to stable storage
2. Output to disk all modified buffer blocks
3. Output to log on stable storage a <**checkpoint** *L*> record.

* Transactions are not allowed to perform any actions while checkpointing is in progress.
* Fuzzy checkpointing allows transactions to progress while the most time consuming parts of checkpointing are in progress

## **Advanced Recovery: Fuzzy Checkpointing**

**Fuzzy checkpointing** is done as follows:

1. Temporarily stop all updates by transactions
2. Write a <**checkpoint** *L*> log record and force log to stable storage
3. Note list *M* of modified buffer blocks
4. Now permit transactions to proceed with their actions
5. Output to disk all modified buffer blocks in list *M*

* blocks should not be updated while being output
* Follow WAL: all log records pertaining to a block must be output before the block is output

1. Store a pointer to the **checkpoint** record in a fixed position **last**\_**checkpoint** on disk

……

<

checkpoint L

>

…..

<

checkpoint L

>

…..

Log

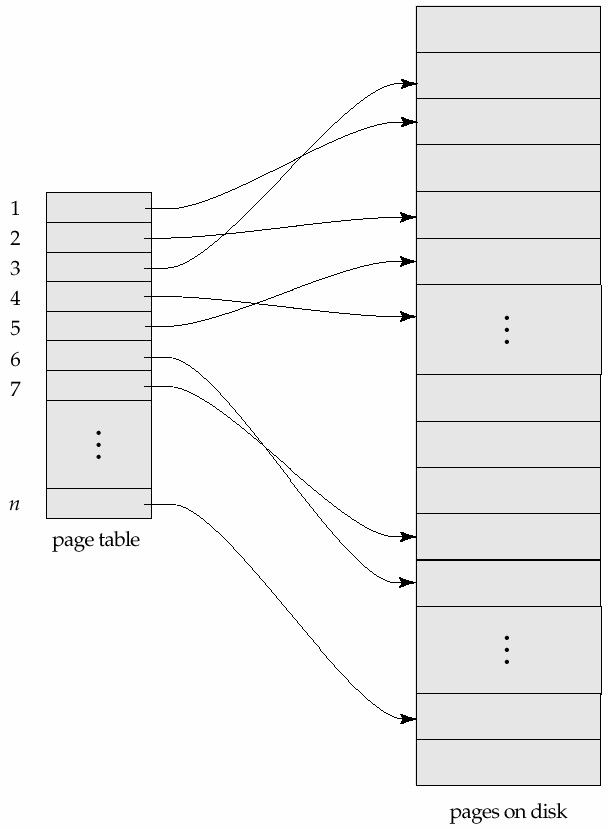
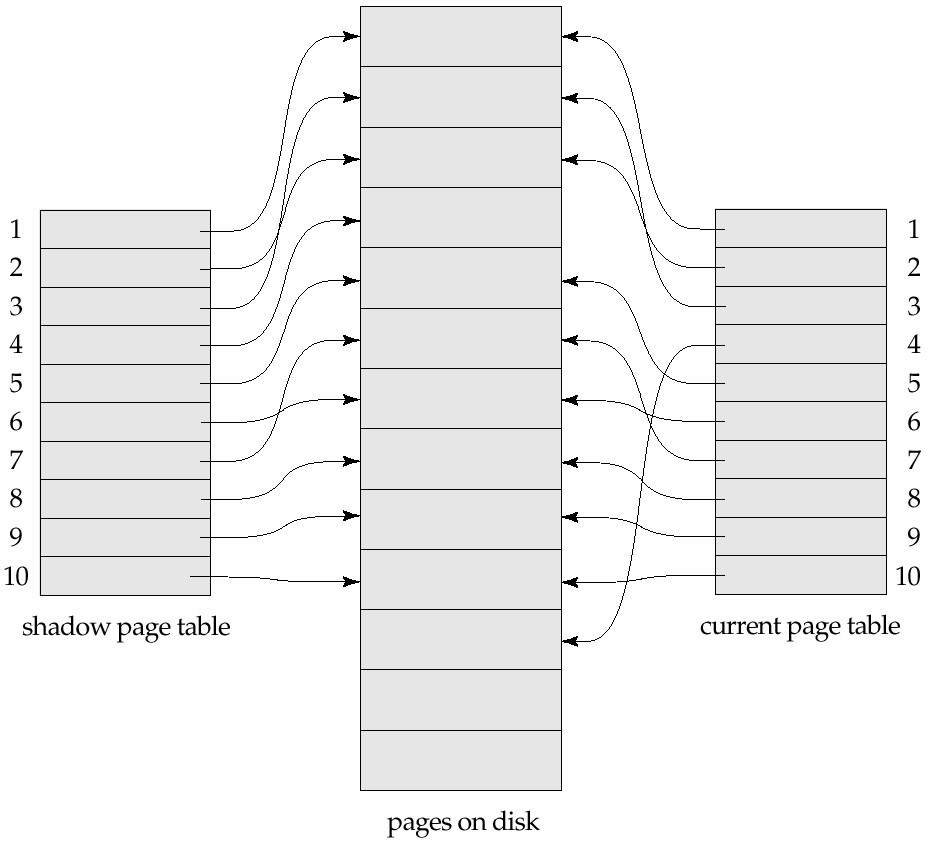
last\_checkpoint

* When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last**\_**checkpoint**
  + Log records before **last**\_**checkpoint** have their updates reflected in database on disk, and need not be redone.
  + Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely

# Shadow Paging

* **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
* Idea: maintain *two* page tables during the lifetime of a transaction –the **current page table**, and the **shadow page table**
* Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
  + Shadow page table is never modified during execution
* To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
* Whenever any page is about to be written for the first time z A copy of this page is made onto an unused page.
  + The current page table is then made to point to the copy
  + The update is performed on the copy

# Sample Page Table



# Example of Shadow Paging: Shadow and current page tables after write to page 4

To commit a transaction:

1. Flush all modified pages in main memory to disk
2. Output current page table to disk
3. Make the current page table the new shadow page table, as follows:

* Keep a pointer to the shadow page table at a fixed (known) location on disk.
* to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
* Once pointer to shadow page table has been written, transaction is committed.
* No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
* Pages not pointed to from current/shadow page table should be freed (garbage collected).

**Advantages of shadow-paging over log-based schemes**

* no overhead of writing log records
* recovery is trivial

**Disadvantages:**

* Copying the entire page table is very expensive
  + Can be reduced by using a page table structured like a B+-tree
  + No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
* Commit overhead is high even with above extension
  + Need to flush every updated page, and page table
  + Data gets fragmented (related pages get separated on disk)
  + After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
* Hard to extend algorithm to allow transactions to run concurrently
  + Easier to extend log based schemes

**Recovery in Multidatabase Systems**

To maintain the atomicity of a multidatabase transaction, it is necessary to have a two-level recovery mechanism. A global recovery manager, or coordinator, is needed to maintain information needed for recovery, in addition to the local recovery managers and information they maintain (log, tables). The coordinator usually follows a protocol called the two-phase commit protocol, whose two phases can be stated as follows:

**• Phase 1:** When all participating databases signal the coordinator that the part of the multidatabase transaction involving each has concluded, the coordinator sends a message "prepare for commit" to each participant to get ready for committing the transaction. Each participating database receiving that message will force-write all log records and needed information for local recovery to disk and then send a "ready to commit" or "OK" signal to the coordinator. If the force-writing to disk fails or the local transaction cannot commit for some reason, the participating database sends a "cannot commit" or "not OK" signal to the coordinator. If the coordinator does not receive a reply from a database within a certain time out interval, it assumes a "not OK" response.

**• Phase 2:** If all participating databases reply "OK," and the coordinator’s vote is also "OK," the transaction is successful, and the coordinator sends a "commit" signal for the transaction to the participating databases. Because all the local effects of the transaction and information needed for local recovery have been recorded in the logs of the participating databases, recovery from failure is now possible. Each participating database completes transaction commit by writing a [commit] entry for the transaction in the log and permanently updating the database if needed. On the other hand, if one or more of the participating databases or the coordinator have a "not OK" response, the transaction has failed, and the coordinator sends a message to "roll back" or UNDO the local effect of the transaction to each participating database. This is done by undoing the transaction operations, using the log.

The net effect of the two-phase commit protocol is that either all participating databases commit the effect of the transaction or none of them do. In case any of the participants—or the coordinator—fails, it is always possible to recover to a state where either the transaction is committed or it is rolled back. A failure during or before Phase 1 usually requires the transaction to be rolled back, whereas a failure during Phase 2 means that a successful transaction can recover and commit.

**REFERENCES**

* Silberschatz, Korth and Sudarshan, Database System Concepts, 5th Edition, Oct 5, 2006
* Fundamentals of Database Systems by Elmasri Navathe