DISTRIBUTED DATABASES (R22CSE3146)

III B.Tech I Semester (Information Technology)

UNIT – IV

Distributed DBMS Reliability : Reliability concepts and measures, fault-tolerance in distributed systems, failures in Distributed DBMS, local & distributed reliability protocols, site failures and network partitioning.

Parallel Database Systems : Parallel database system architectures, parallel data placement, parallel query processing, load balancing, database clusters.

Distributed DBMS Reliability

- A **reliable DDBMS** is one that can continue to process user requests even when the underlying system is unreliable, i.e., failures occur.

  Data replication + Easy scaling = **Reliable system**
- **Distribution** enhances system reliability (not enough). Need number of protocols to be implemented to exploit distribution and replication.
- Reliability is closely related to the problem of how to maintain the **atomicity** and **durability** properties of transactions.

Reliability Concepts and Measures

- A reliable distributed database management system is one that can continue to process user requests even when the underlying system is unreliable.
- When components of the distributed computing environment fail, a reliable distributed DBMS should be able to continue executing user requests without violating database consistency.

1. System, State, and Failure

- Reliability refers to a ***system*** that consists of a set of ***components.***
- The system has a ***state***, which changes as the system operates.
- The behavior of the system in providing response to all the possible external stimuli is laid out in an authoritative ***specification*** of its behavior.
- The specification indicates the valid behavior of each system state.
- Any deviation of a system from the behavior described in the specification is considered a ***failure.***
- For example, in a distributed transaction manager the specification may state that only serializable schedules for the execution of concurrent transactions should be generated.
- If the transaction manager generates a non-serializable schedule, we say that it has **failed.**
- Each failure obviously needs to be traced back to its cause.

- Failures in a system can be attributed to deficiencies either in the components that make it up, or in the design, that is, how these components are put together.
- Each state that a reliable system goes through is valid in the sense that the state fully meets its specification.
- However, in an unreliable system, it is possible that the system may get to an internal state that may not obey its specification.
- Further transitions from this state would eventually cause a system **failure.**
- Such internal states are called ***erroneous states***; the part of the state that is incorrect is called an ***error*** in the system.
- Any error in the internal states of the components of a system or in the design of a system is called a ***fault*** in the system.
- Thus, a fault causes an error that results in a system failure.

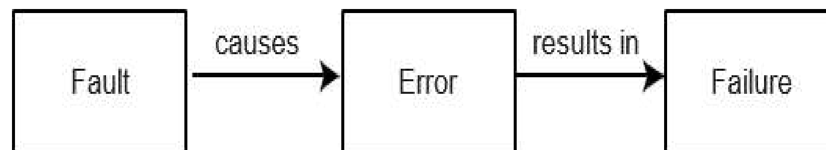

Fig. Chain of Events Leading to System Failure

- We differentiate between errors (or faults and failures) that are permanent and those that are not permanent. (Types of Faults)

Hard faults

- Permanent
- Resulting failures are called hard failures
- Permanence can apply to a failure, a fault, or an error, although we typically use the term with respect to faults.
- A *permanent fault*, also commonly called a *hard fault*, is one that reflects an irreversible change in the behavior of the system.
- Permanent faults cause permanent errors that result in permanent failures.
- The characteristics of these failures is that recovery from them requires intervention to "repair" the fault.

Soft faults.

- Transient or intermittent
- Account for more than 90% of all failures
- Resulting failures are called soft failures
- An intermittent fault refers to a fault that demonstrates itself occasionally due to unstable hardware or varying hardware or software states.
- A typical example is the faults that systems may demonstrate when the load becomes too heavy.
- On the other hand, a transient fault describes a fault that results from temporary environmental conditions.
- A transient fault might occur, for example, due to a sudden increase in the room

temperature.

- The transient fault is therefore the result of environmental conditions that may be impossible to repair.
- An intermittent fault, on the other hand, can be repaired since the fault can be traced to a component of the system.
- Remember that we have also indicated that system failures can be due to design faults. Design faults together with unstable hardware cause intermittent errors that result in system failure.
- A final source of system failure that may not be attributable to a component fault or a design fault is operator mistakes.

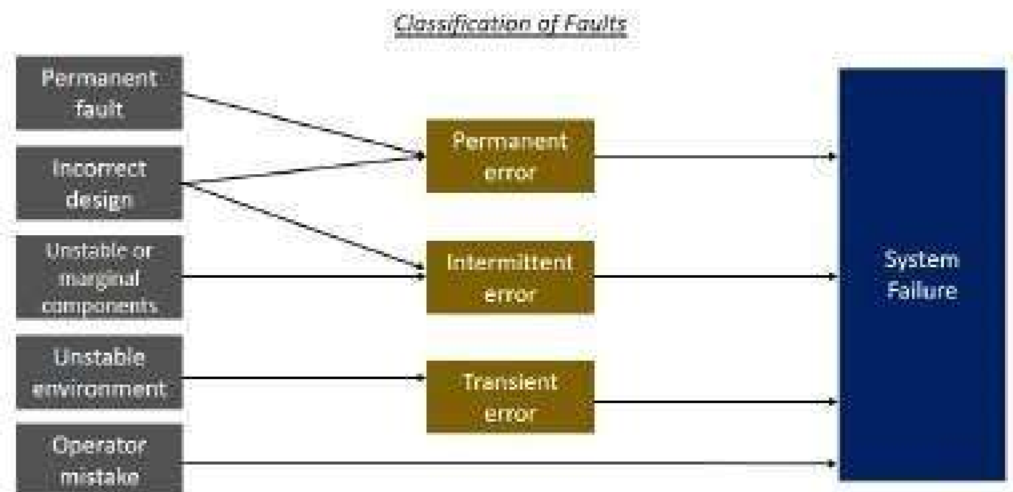

Fig : Sources of System Failure

2. **Reliability and Availability**

- **Reliability**:
  - A measure of success with which a system conforms to some authoritative specification of its behavior.
  - Probability that the system has not experienced any failures within a given time period.

- **Availability**:
  - The fraction of the time that a system meets its specification.
  - The probability that the system is operational at a given time *t*.

- ***Reliability*** refers to the probability that the system under consideration does not experience any failures in a given time interval. It is typically used to describe systems that cannot be repaired (as in space-based computers), or where the operation of the system is so critical that no downtime for repair can be tolerated.
- Formally, the reliability of a system, $R(t)$, is defined as the following conditional probability:

$$R(t) = \Pr\{0 \text{ failures in time } [0,t] | \text{no failures at } t = 0\}$$

- If we assume that failures follow a Poisson distribution (which is usually the case for hardware), this formula reduces to

$$R(t) = \Pr\{0 \text{ failures in time } [0,t]\}$$

- Under the same assumptions, it is possible to derive that

$$\Pr\{k \text{ failures in time } [0,\ t]\} = \frac{e^{-m(t)}[m(t)]^k}{k!}$$

- where $m(t) = \int_0^t z(x)\,dx$. Here $z(t)$ is known as the *hazard function*, which gives the time-dependent failure rate of the specific hardware component under consideration.
- The probability distribution for $z(t)$ may be different for different electronic components.
  The expected (mean) number of failures in time $[0, t]$ can then be computed as

$$E[k] = \sum_{k=0}^{\infty} k \frac{e^{-m(t)}[m(t)]^k}{k!} = m(t)$$

and the variance as

$$Var[k] = E[k^2] - (E[k])^2 = m(t)$$

Given these values, $R(t)$ can be written as

$$R(t) = e^{-m(t)}$$

- Note that the reliability equation above is written for one component of the system. For a system that consists of n non-redundant components (i.e., they all have to function properly for the system to work) whose failures are independent, the overall system reliability can be written as

$$R_{sys}(t) = \Pi_{i=1}^{n} R_i(t)$$

- ***Availability*, *A*(*t*),** refers to the probability that the system is operational according to its specification at a given point in time t.
- A number of failures may have occurred prior to time t, but if they have all been repaired, the system is available at time t. Obviously, availability refers to systems that can be repaired.
- If one looks at the limit of availability as time goes to infinity, it refers to the expected percentage of time that the system under consideration is available to perform useful computations.
- Availability can be used as some measure of "goodness" for those systems that can be repaired and which can be out of service for short periods of time during repair.
- Reliability and availability of a system are considered to be contradictory objectives.
- It is usually accepted that it is easier to develop highly available systems as opposed to highly reliable systems.
- If we assume that failures follow a Poisson distribution with a failure rate λ, and that repair time is exponential with a mean repair time of $1/\mu$, the steady-state availability of a system can be written as

$$A = \frac{\mu}{\lambda + \mu}$$

NRCM
Your roots to success...

3. Mean Time between Failures/Mean Time to Repair

- Two single-parameter measures have become more popular than the reliability and availability functions given above to model the behavior of systems.
- These two measures used are
- *Mean Time Between Failures* (MTBF)
- *Mean Time To Repair* (MTTR).
- MTBF is the expected time between subsequent failures in a system with repair.
- MTBF can be calculated either from empirical data or from the reliability function as

$$\text{MTBF} = \int_0^\infty R(t)\,dt$$

- Since $R(t)$ is related to the system failure rate, there is a direct relationship between MTBF and the failure rate of a system.
- MTTR is the expected time to repair a failed system. It is related to the repair rate as MTBF is related to the failure rate.
- Using these two metrics, the steady-state availability of a system with exponential failure and repair rates can be specified as

$$A = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

- System failures may be *latent*, in that a failure is typically detected some time after its occurrence.
- This period is called *error latency*, and the average error latency time over a number of identical systems is called *mean time to detect* (MTTD).

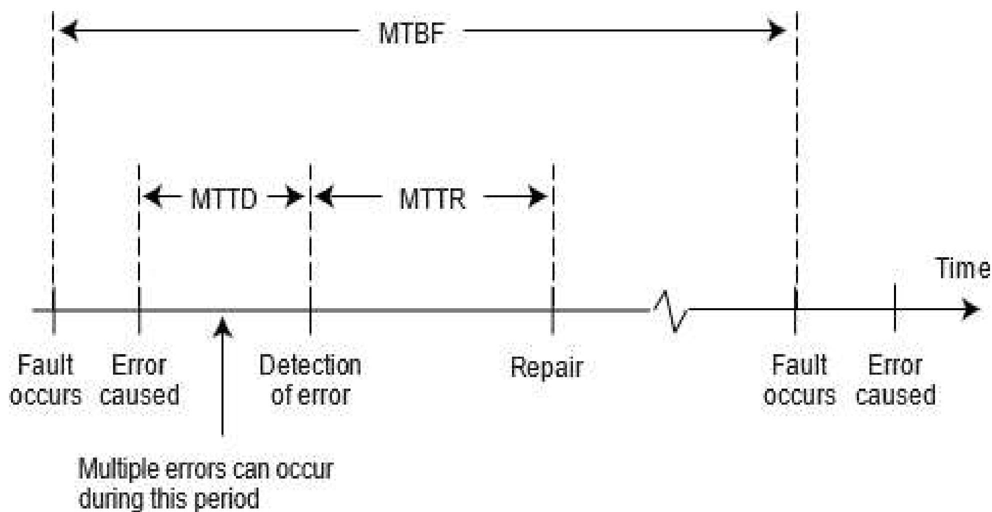

Fig. Occurrence of Events over Time

Fault-Tolerance in Distributed Systems

In distributed systems, three types of problems occur. All these three types of problems are related.

- **Fault:** Fault is defined as a weakness or shortcoming in the system or any hardware and software component. The presence of fault can lead to error and failure.
- **Errors:** Errors are incorrect results due to the presence of faults.
- **Failure:** Failure is the outcome where the assigned goal is not achieved.

Types of Faults

There are three types of Faults :

1. **Transient Faults**
2. **Intermittent Faults**
3. **Permanent Faults**

Transient Faults:

- Transient Faults are the type of faults that occur once and then disappear.
- These types of faults do not harm the system to a great extent but are very difficult to find or locate.
- Processor fault is an example of transient fault.

Intermittent Faults:

- Intermittent Faults are the type of faults that come again and again.
- Such as once the fault occurs it vanishes upon itself and then reappears again.
- An example of intermittent fault is when the working computer hangs up.

Permanent Faults:

- Permanent Faults are the type of faults that remain in the system until the component is replaced by another.
- These types of faults can cause very severe damage to the system but are easy to identify.
- A burnt-out chip is an example of a permanent Fault.

What is Fault Tolerance?

- Fault Tolerance is defined as the ability of the system to function properly even in the presence of any failure.
- Distributed systems consist of multiple components due to which there is a high risk of faults occurring.
- Due to the presence of faults, the overall performance may degrade.

Need for Fault Tolerance in Distributed Systems

Fault Tolerance is required in order to provide below four features.

1. **Availability:** Availability is defined as the property where the system is readily available for its use at any time.

2. **Reliability:** Reliability is defined as the property where the system can work continuously without any failure.

3. **Safety:** Safety is defined as the property where the system can remain safe from unauthorized access even if any failure occurs.

4. **Maintainability:** Maintainability is defined as the property states that how easily and fastly the failed node or system can be repaired.

Phases of Fault Tolerance in Distributed Systems

- In order to implement the techniques for fault tolerance in distributed systems, the design, configuration and relevant applications need to be considered.
- Below are the phases carried out for fault tolerance in any distributed systems.

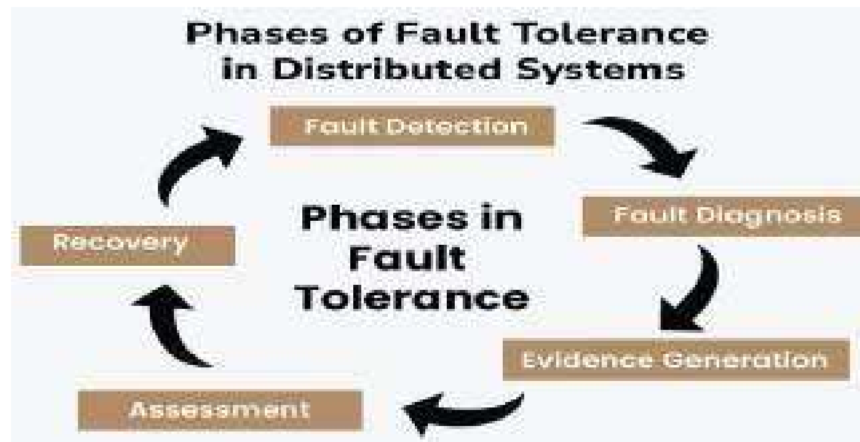

Fig : Phases of Fault Tolerance in Distributed Systems

1. Fault Detection

- Fault Detection is the first phase where the system is monitored continuously.
- The outcomes are being compared with the expected output.
- During monitoring if any faults are identified they are being notified.
- These faults can occur due to various reasons such as hardware failure, network failure, and software issues.
- The main aim of the first phase is to detect these faults as soon as they occur so that the work being assigned will not be delayed.

2. Fault Diagnosis

- Fault diagnosis is the process where the fault that is identified in the first phase will be diagnosed properly in order to get the root cause and possible nature of the faults.

- Fault diagnosis can be done manually by the administrator or by using automated Techniques in order to solve the fault and perform the given task.

3. Evidence Generation

- Evidence generation is defined as the process where the report of the fault is prepared based on the diagnosis done in an earlier phase.
- This report involves the details of the causes of the fault, the nature of faults, the solutions that can be used for fixing, and other alternatives and preventions that need to be considered.

4. Assessment

- Assessment is the process where the damages caused by the faults are analyzed.
- It can be determined with the help of messages that are being passed from the component that has encountered the fault.
- Based on the assessment further decisions are made.

5. Recovery

- Recovery is the process where the aim is to make the system fault free.
- It is the step to make the system fault free and restore it to state forward recovery and backup recovery.
- Some of the common recovery techniques such as reconfiguration and resynchronization can be used.

Types of Fault Tolerance in Distributed Systems

your roots to success...

1. **Hardware Fault Tolerance:**
   - Hardware Fault Tolerance involves keeping a backup plan for hardware devices such as memory, hard disk, CPU, and other hardware peripheral devices.
   - Hardware Fault Tolerance is a type of fault tolerance that does not examine faults and runtime errors but can only provide hardware backup.
   - The two different approaches that are used in Hardware Fault Tolerance are fault-masking and dynamic recovery.

2. **Software Fault Tolerance:**
   - Software Fault Tolerance is a type of fault tolerance where dedicated software is used in order to detect invalid output, runtime, and programming errors.
   - Software Fault Tolerance makes use of static and dynamic methods for detecting and providing the solution.
   - Software Fault Tolerance also consists of additional data points such as recovery rollback and checkpoints.

3. **System Fault Tolerance:**
   - System Fault Tolerance is a type of fault tolerance that consists of a whole system.
   - It has the advantage that it not only stores the checkpoints but also the memory block, and

program checkpoints and detects the errors in applications automatically.

- If the system encounters any type of fault or error it does provide the required mechanism for the solution.
- Thus system fault tolerance is reliable and efficient.

Failures in Distributed DBMS

- Designing a reliable system that can recover from failures requires identifying the types of failures with which the system has to deal.
- In a distributed database system, we need to deal with four types of failures:
    1. Transaction failures (aborts)
    2. Site (system) failures
    3. Media (disk) failures
    4. Communication line failures

1. Transaction Failures (Aborts)

- Transactions can fail for a number of reasons.
- Failure can be due to an error in the transaction caused by incorrect input data as well as the detection of a present or potential deadlock.
- Some concurrency control algorithms do not permit a transaction to proceed or even to wait if the data that they attempt to access are currently being accessed by another transaction. This might also be considered a failure.
- The usual approach to take in cases of transaction failure is to *abort* the transaction, thus resetting the database to its state prior to the start of this transaction.

2. Site (System) Failures

- The reasons for system failure can be traced back to a hardware or to a software failure.
- The system failure is always assumed to result in the loss of main memory contents.
- Therefore, any part of the database that was in main memory buffers is lost as a result of a system failure.
- The database that is stored in secondary storage is assumed to be safe and correct.
- In distributed database terminology, system failures are typically referred to as *site failures*, since they result in the failed site being unreachable from other sites in the distributed system.
- We typically differentiate between partial and total failures in a distributed system.
- *Total failure* refers to the simultaneous failure of all sites in the distributed system;
- *partial failure* indicates the failure of only some sites while the others remain operational.

3. Media (Disk) Failures

- *Media failure* refers to the failures of the secondary storage devices that store the database.
- Such failures may be due to operating system errors, as well as to hardware faults such as head crashes or controller failures.

- The important point from the perspective of DBMS reliability is that all or part of the database that is on the secondary storage is considered to be destroyed and inaccessible.
- Media failures are frequently treated as problems local to one site and therefore not specifically addressed in the reliability mechanisms of distributed DBMSs.

4. Communication line failures

- The three types of failures described above are common to both centralized and distributed DBMSs.
- Communication failures are unique to the distributed case.
- There are a number of types of communication failures.
- The most common ones are the errors in the messages, improperly ordered messages, lost (or undeliverable) messages, and communication line failures.
- If a communication line fails, in addition to losing the message(s) in transit, it may also divide the network into two or more disjoint groups.
- This is called ***network partitioning***. If the network is partitioned, the sites in each partition may continue to operate.
- Executing transactions that access data stored in multiple partitions becomes a major issue.
- The term for the failure of the communication network to deliver messages and the confirmations within this period is ***performance failure***.
- It needs to be handled within the reliability protocols for distributed DBMSs.

Local & Distributed Reliability Protocols

Local Reliability Protocols

- Here, we discuss the functions performed by the **Local Recovery Manager** (LRM) that exists at each site.These functions maintain the atomicity and durability properties of local transactions.
- They relate to the execution of the commands that are passed to the LRM, which are **begin transaction**, **read**, **write**, **commit**, and **abort**.
- When the LRM wants to read a page of data on behalf of a transaction it uses a fetch command, indicating the page that it wants to read.

1. Architectural Considerations

Volatile database

- The database buffer manager keeps some of the recently accessed data in main memory buffers.
- This is done to enhance access performance.
- Typically, the buffer is divided into pages that are of the same size as the stable database pages.
- The part of the database that is in the database buffer is called the ***volatile database***.
- It is important to note that the LRM executes the operations on behalf of a transaction only on the

volatile database, which, at a later time, is written back to the stable database.

Stable database

- We assume that the database is stored permanently on secondary storage, which in this context is called the *stable storage*.
- The stability of this storage medium is due to its robustness to failures.
- A stable storage device would experience considerably less-frequent failures than would a non-stable storage device.
- In today's technology, stable storage is typically implemented by means of duplexed magnetic disks which store duplicate copies of data that are always kept mutually consistent (i.e., the copies are identical).
- We call the version of the database that is kept on stable storage the ***stable database***.
- The unit of storage and access of the stable database is typically a *page*.

Buffer Manager

- The buffer manager acts as a conduit for all access to the database via the buffers that it manages.
- It provides this function by fulfilling three tasks:

1. *Searching* the buffer pool for a given page;
2. If it is not found in the buffer, *allocating* a free buffer page and *loading* the buffer page with a data page that is brought in from secondary storage;
3. If no free buffer pages are available, choosing a buffer page for *replacement*.

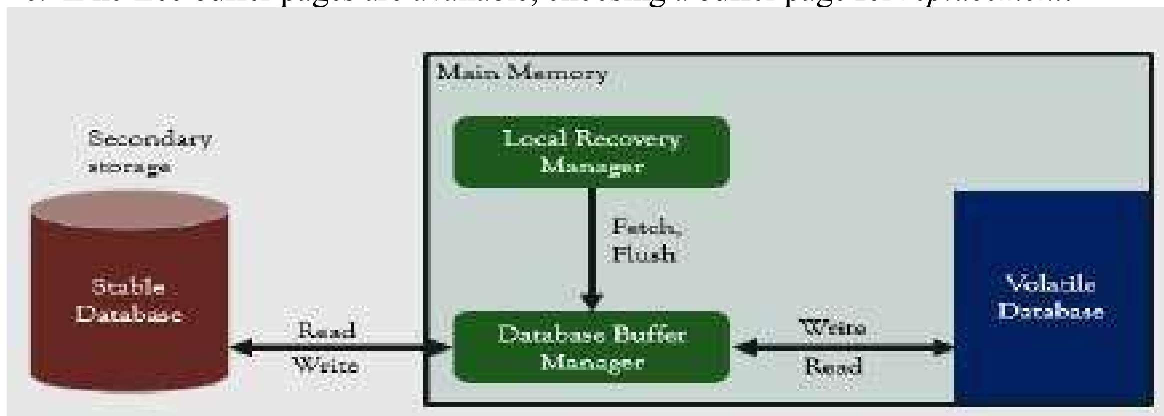

Fig. Interface Between the Local Recovery Manager and the Buffer Manager

2. Recovery Information

- When a system failure occurs, the volatile database is lost.
- Therefore, the DBMS has to maintain some information about its state at the time of the failure in order to be able to bring the database to the state that it was in when the failure occurred. We call this information the ***recovery information***.

- The recovery information that the system maintains is dependent on the method of executing updates.
- Two possibilities are in-place updating and out-of-place updating.
- ***In-place updating*** physically changes the value of the data item in the stable database. As a result, the previous values are lost.
- ***Out-of-place updating*** does not change the value of the data item in the stable database but maintains the new value separately.

In-Place Update Recovery Information

Logging:

- Since in-place updates cause previous values of the affected data items to be lost, it is necessary to keep enough information about the database state changes to facilitate the recovery of the database to a consistent state following a failure.
- This information is typically maintained in a ***database log***.
- Thus each update transaction not only changes the database but the change is also recorded in the database log.
- The log contains information necessary to recover the database state following a failure.

Fig. Update Operation Execution

- Assume that the LRM and buffer manager algorithms are such that the buffer pages are written back to the stable database only when the buffer manager needs new buffer space.
- The **flush** command is not used by the LRM and the decision to write back the pages into the stable database is taken at the discretion of the buffer manager.
- Now consider that a transaction T_1 had completed (i.e., committed) before the failure occurred.
- The durability property of transactions would require that the effect os T_1 be reflected in the database.
- It is possible that the volatile database pages that have been updated by T_1 may not have been written back to the stable database at the time of the failure.
- Therefore, upon recovery, it is important to be able to *redo* the operations of T_1.
- This requires some information to be stored in the database log about the effects of T_1.
- Given this information, it is possible to recover the database from its “old” state to the “new” state that reflects the effects of T_1.

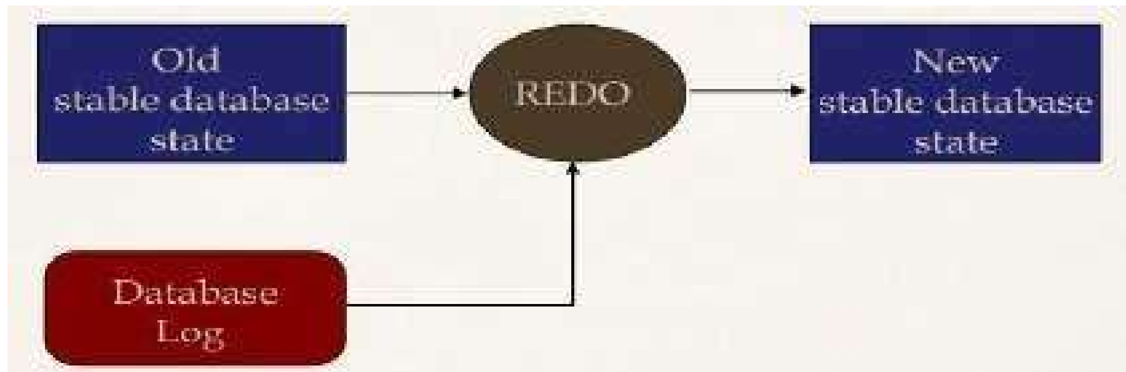

Fig. REDO Action

- Now consider another transaction, T_2, that was still running when the failure occurred.
- The atomicity property would dictate that the stable database not contain any effects of T_2.
- It is possible that the buffer manager may have had to write into the stable database some of the volatile database pages that have been updated by T_2.
- Upon recovery from failures it is necessary to *undo* the operations of T_2.
- Thus the recovery information should include sufficient data to permit the undo by taking the "new" database state that reflects partial effects of T_2 and recovers the "old" state that existed at the start of T_2.
- We should indicate that the undo and redo actions are assumed to be idempotent.
- Their repeated application to a transaction would be equivalent to performing them once.
- The undo/redo actions form the basis of different methods of executing the commit commands.

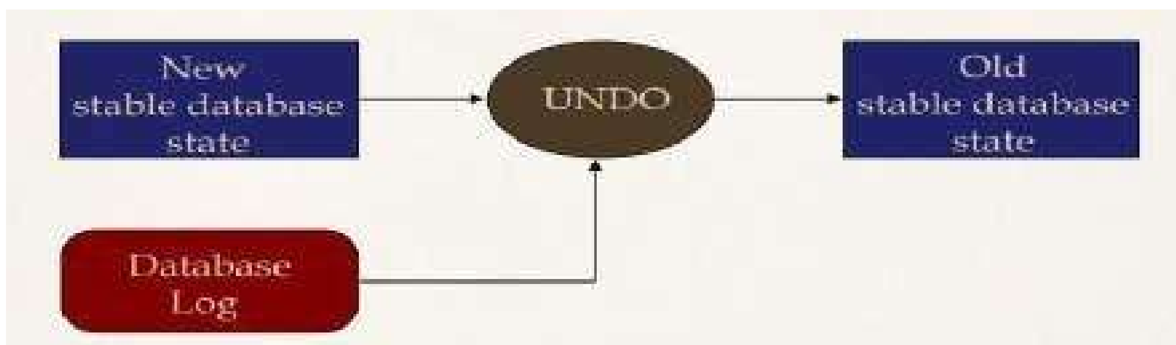

Fig. UNDO Action

- The contents of the log may differ according to the implementation.
- The following minimal information for each transaction is contained in almost all database logs:
- A begin transaction record, the value of the data item before the update (called the *before image*)
- The updated value of the data item (called the *after image*)
- A termination record indicating the transaction termination condition (commit, abort).
- The granularity of the before and after images may be different, as it is possible to log entire pages or some smaller unit.
- As an alternative to this form of *state logging*, *operational logging*, may be supported where the operations that cause changes to the database are logged rather than the before and after images.

- The log is also maintained in main memory buffers (called *log buffers*) and written back to stable storage (called *stable log*) similar to the database buffer pages.
- The log pages can be written to stable storage in one of two ways.
- They can be written *synchronously* (more commonly known as *forcing a log*) where the addition of each log record requires that the log be moved from main memory to stable storage.
- They can also be written *asynchronously*, where the log is moved to stable storage either at periodic intervals or when the buffer fills up.
- When the log is written synchronously, the execution of the transaction is suspended until the write is complete. This adds some delay to the response-time performance of the transaction.
- On the other hand, if a failure occurs immediately after a forced write, it is relatively easy to recover to a consistent database state.

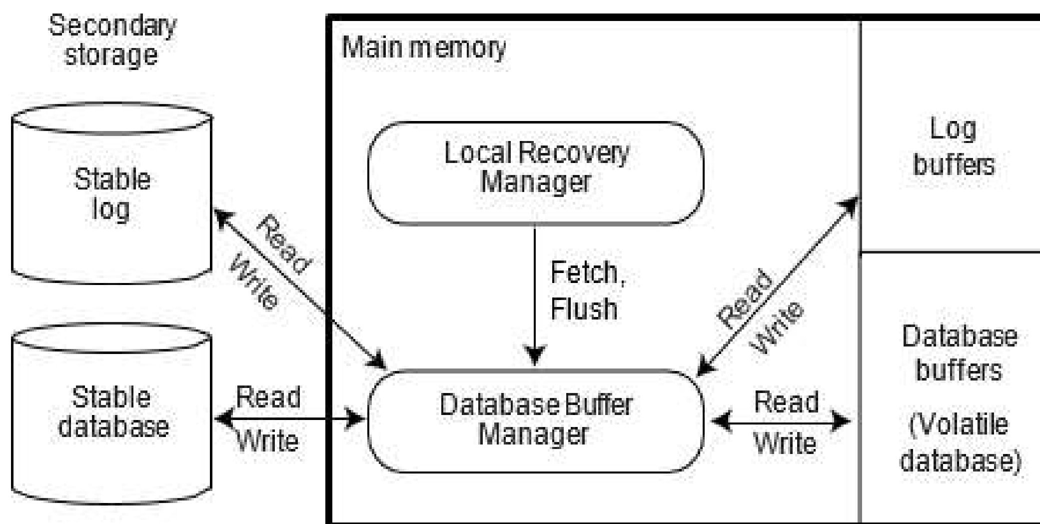

Fig. Logging Interface

- Whether the log is written synchronously or asynchronously, one very important protocol has to be observed in maintaining logs.
- Consider a case where the updates to the database are written into the stable storage before the log is modified in stable storage to reflect the update.
- If a failure occurs before the log is written, the database will remain in updated form, but the log will not indicate the update that makes it impossible to recover the database to a consistent and up-to-date state. Therefore, the stable log is always updated prior to the updating of the stable database.
- This is known as the *write-ahead logging* (WAL) protocol and can be precisely specified as follows:

Write-Ahead Log protocol

1. Before a stable database is updated (perhaps due to actions of a yet uncommitted transaction), the before images should be stored in the stable log. This facilitates undo.
2. When a transaction commits, the after images have to be stored in the stable log prior to the updating of the stable database. This facilitates redo.

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).

Example for in-place update

- Example: Consider the transactions T_0 and T_1 (T_0 executes before T_1) and the following initial values: A=1000, B=2000, and C=700

- T_0: Read(A)
  A=A-50
  Write(A)
  Read(B)
  B=B+50
  Write(B)

- T_1: Read(C)
  C=C-100
  Write(C)

Possible order of actual outputs to the log file and the DB:

Log	DB
<T_0,start>	
<T_0,A,1000,950>	
<T_0,B,2000,2050>	
<T_0,commit>	
	A=950 B=2050
<T_1,start>	
<T_1,C,700,600>	
<T_1,commit>	
	C=600

Example continued..

- Consider the log after some system crashes and the corresponding recovery actions

(a) <T_0,start>
<T_0,A,1000,950>
<T_0,B,2000,2050>

(b) <T_0,start>
<T_0,A,1000,950>
<T_0,B,2000,2050>
<T_0,commit>
<T_1,start>
<T_1,C,700,600)

(c) <T_0,start>
<T_0,A,1000,950>
<T_0,B,2000,2050>
<T_0,commit>
<T_1,start>
<T_1,C,700,600)
<T_1,commit>

(a). undo(T_0): B is restored to 2000 and A to 1000

(b). Undo(T_1) and redo(T_0): C is restored to 700, and then A and B are set to 950 and 2050, respectively

(c). Redo(T_0) and redo (T_1): And B are set to 950 and 2050, respectively; then C is set to 600

Out-of-Place Update Recovery Information

Shadowing:

- Typical techniques for out-of-place updating are *shadowing* and *differential files*.
- Shadowing uses duplicate stable storage pages in executing updates.
- Thus every time an update is made, the old stable storage page, called the *shadow page*, is left intact and a new page with the updated data item values is written into the stable database.
- The access path data structures are updated to point to the new page, which contains the current data so that subsequent accesses are to this page.
- The old stable storage page is retained for recovery purposes (to perform undo).
- Recovery based on shadow paging is implemented in System R's recovery manager.
- This implementation uses shadowing together with logging.

The Differential File

- The method maintains each stable database file as a read-only file.
- In addition, it maintains a corresponding read-write differential file that stores the changes to that file.
- Given a logical database file F, let us denote its read-only part as FR and its corresponding differential file as DF.
- DF consists of two parts:
- Insertions part, which stores the insertions to F, denoted DF^+.
- Deletions part, denoted DF^-.
- All updates are treated as the deletion of the old value and the insertion of a new one.
- Thus each logical file F is considered to be a view defined as $F = (FR \cup DF^+) - DF^-$.
- Periodically, the differential file needs to be merged with the read-only base file.

3. Execution of LRM Commands

NRCM your roots to success

- Recall that there are five commands that form the interface to the LRM.
- These are the **begin transaction, read, write, commit**, and **abort** commands.
- In this section we introduce a sixth interface command to the LRM: **recover**.
- The **recover** command is the interface that the operating system has to the LRM.
- It is used during recovery from system failures when the operating system asks the DBMS to recover the database to the state that existed when the failure occurred.
- Now we present the execution methods of the **begin transaction, read**, **write, commit** and **recovery** commands.

Begin transaction, Read, and Write Commands

Begin transaction.

- Assume that it causes the LRM to write a begin transaction record into the log.

- This is an assumption made for convenience of discussion; in reality, writing of the begin transaction record may be delayed until the first **write** to improve performance by reducing I/O.

Read.

- The **read** command specifies a data item.
- The LRM tries to read the specified data item from the buffer pages that belong to the transaction.
- If the data item is not in one of these pages, it issues a **fetch** command to the buffer manager in order to make the data available.
- Upon reading the data, the LRM returns it to the scheduler.

Write.

- The **write** command specifies the data item and the new value.
- As with a read command, if the data item is available in the buffers of the transaction, its value is modified in the database buffers (i.e., the volatile database).
- If it is not in the private buffer pages, a **fetch** command is issued to the buffer manager, and the data is made available and updated.
- The before image of the data page, as well as its after image, are recorded in the log.
- The local recovery manager then informs the scheduler that the operation has been completed successfully.

Execution strategies

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

This type of LRM algorithm is called a redo/undo algorithm since it requires performing both the redo and undo operations upon recovery.

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

The LRM algorithms that use this strategy are called undo/no-redo.

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

In this case the LRM controls the writing of the volatile database pages into stable storage. This precludes the need for a global undo operation and is therefore called a redo/no-undo.

Abort

- None of the updated pages have been written into stable database
- Release the fixed 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 **fixed**

Recover

- Perform partial redo
- No need to perform global undo

Fix/Flush

This is the case where the LRM forces the buffer manager to write the updated volatile database pages into the stable database at precisely the commit point—not before and not after. This strategy is called no-undo/no-redo.

Abort

- None of the updated pages have been written into stable database
- Release the fixed pages

Commit (the following must 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 **fixed**
- LRM writes an “end_of_transaction” record into the log.

Recover

- No need to do anything

4. Checkpointing

- In most of the LRM implementation strategies, the execution of the recovery action requires searching the entire log.
- This is a significant overhead because the LRM is trying to find all the transactions that need to be undone and redone.
- The overhead can be reduced if it is possible to build a wall which signifies that the database at that point is up-to-date and consistent.
- In that case, the redo has to start from that point on and the undo only has to go back to that point. This process of building the wall is called *checkpointing*.
- Checkpointing is achieved in three steps :

    1. Write a begin checkpoint record into the log.
    2. Collect the checkpoint data into the stable storage.
    3. Write an end checkpoint record into the log.

- The first and the third steps enforce the atomicity of the checkpointing operation.
- If a system failure occurs during check pointing, the recovery process will not find an end checkpoint record and will consider checkpointing not completed.
- There are a number of different alternatives for the data that is collected in Step 2, how it is collected, and where it is stored.
- We will consider one example here, called *transaction-consistent checkpointing*.
- The checkpointing starts by writing the begin checkpoint record in the log and stopping the acceptance of any new transactions by the LRM.
- Once the active transactions are all completed, all the updated volatile database pages are flushed to the stable database followed by the insertion of an end checkpoint record into the log.
- In this case, the redo action only needs to start from the end checkpoint record in the log.
- The undo action can go the reverse direction, starting from the end of the log and stopping at the end checkpoint record.
- Transaction-consistent checkpointing is not the most efficient algorithm, since a significant delay is experienced by all the transactions.
- There are alternative check- pointing schemes such as action-consistent checkpoints, fuzzy checkpoints, and others.

5. Handling Media Failures

- As we mentioned before, the previous discussion on centralized recovery considered non-media failures, where the database as well as the log stored in the stable storage survive the failure.
- Media failures may either be quite catastrophic, causing the loss of the stable database or of the stable log, or they can simply result in partial loss of the database or the log.
- The methods that have been devised for dealing with this situation are again based on duplexing.

- To cope with catastrophic media failures, an *archive* copy of both the database and the log is maintained on a different (tertiary) storage medium, which is typically the magnetic tape or CD-ROM.
- Thus the DBMS deals with three levels of memory hierarchy: the main memory, random access disk storage, and magnetic tape.
- To deal with less catastrophic failures, having duplicate copies of the database and log may be sufficient.
- When a media failure occurs, the database is recovered from the archive copy by redoing and undoing the transactions as stored in the archive log.
- The real question is how the archive database is stored.
- If we consider the large sizes of current databases, the overhead of writing the entire database to tertiary storage is significant.
- Two methods that have been proposed for dealing with this are
  - To perform the archiving activity concurrent with normal processing
  - To archive the database incrementally as changes occur so that each archive version contains only the changes that have occurred since the previous archiving.

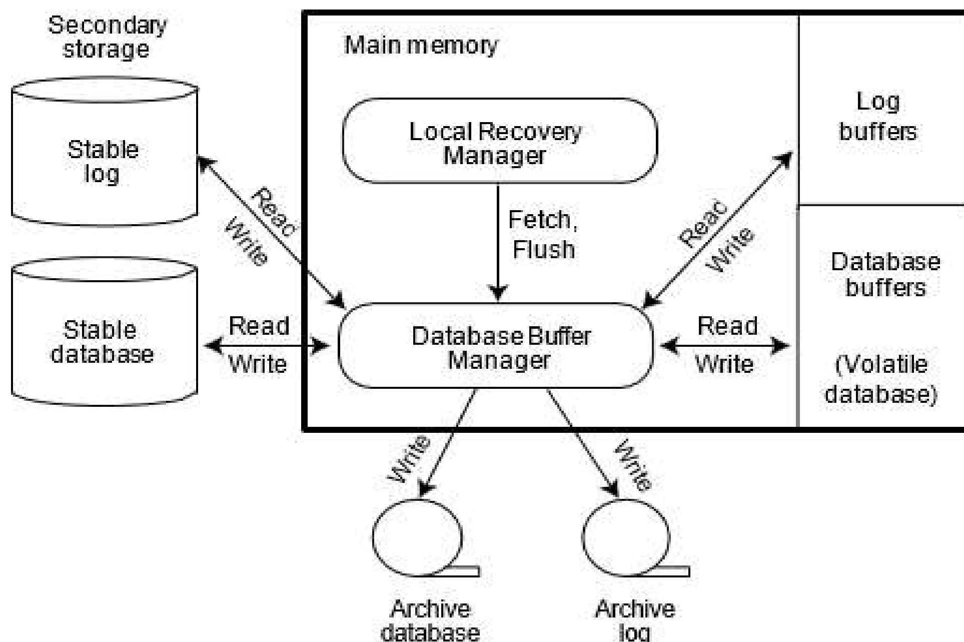

Fig : Full Memory Hierarchy Managed by LRM and BM

Distributed Reliability Protocols

- The distributed versions aim to maintain the atomicity and durability of distributed transactions that execute over a number of databases.
- The protocols address the distributed execution of the **begin transaction, read, write, abort, commit**, and **recover** commands.
- The distributed reliability protocols are implemented between the coordinator and the participants.

They are :
1. Components of Distributed Reliability Protocols
2. Two-Phase Commit Protocol

1. Components of Distributed Reliability Protocols:

- Reliability techniques consist of commit, termination, and recovery protocols
- Commit and recover commands executed differently in a distributed DBMS than centralized
- Termination protocols are unique to distributed systems
- Termination vs. recovery protocols
    - Opposite faces of recovery problem
    - Given a site failure, termination protocols address how the operational sites deal with the failure
    - Recovery protocols address procedure the process at the failed site must go through to recover its state
- Commit protocols must maintain atomicity of distributed transactions
- Ideally recovery protocols are independent – no need to consult other sites to terminate transaction

2. **Two-Phase Commit Protocol**

- Insists that all sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent
- Scheduler issues, deadlocks necessitate synchronization
- Global commit rule:
    - If even one participant votes to abort the transaction, the coordinator must reach a global abort decision
    - If all the participants vote to commit the transaction, the coordinator must reach a global commit decision
- A participant may unilaterally abort a transaction until an affirmative vote is registered
- A participant's vote cannot be changed
- Timers used to exit from wait states

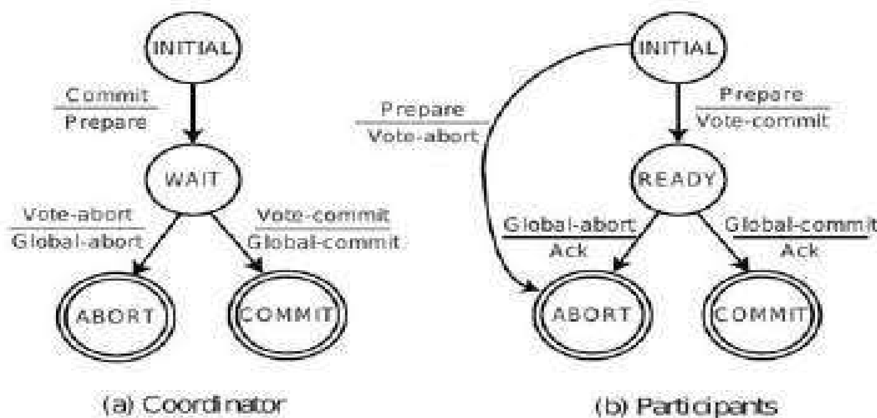

Fig : State Transitions in 2PC Protocol

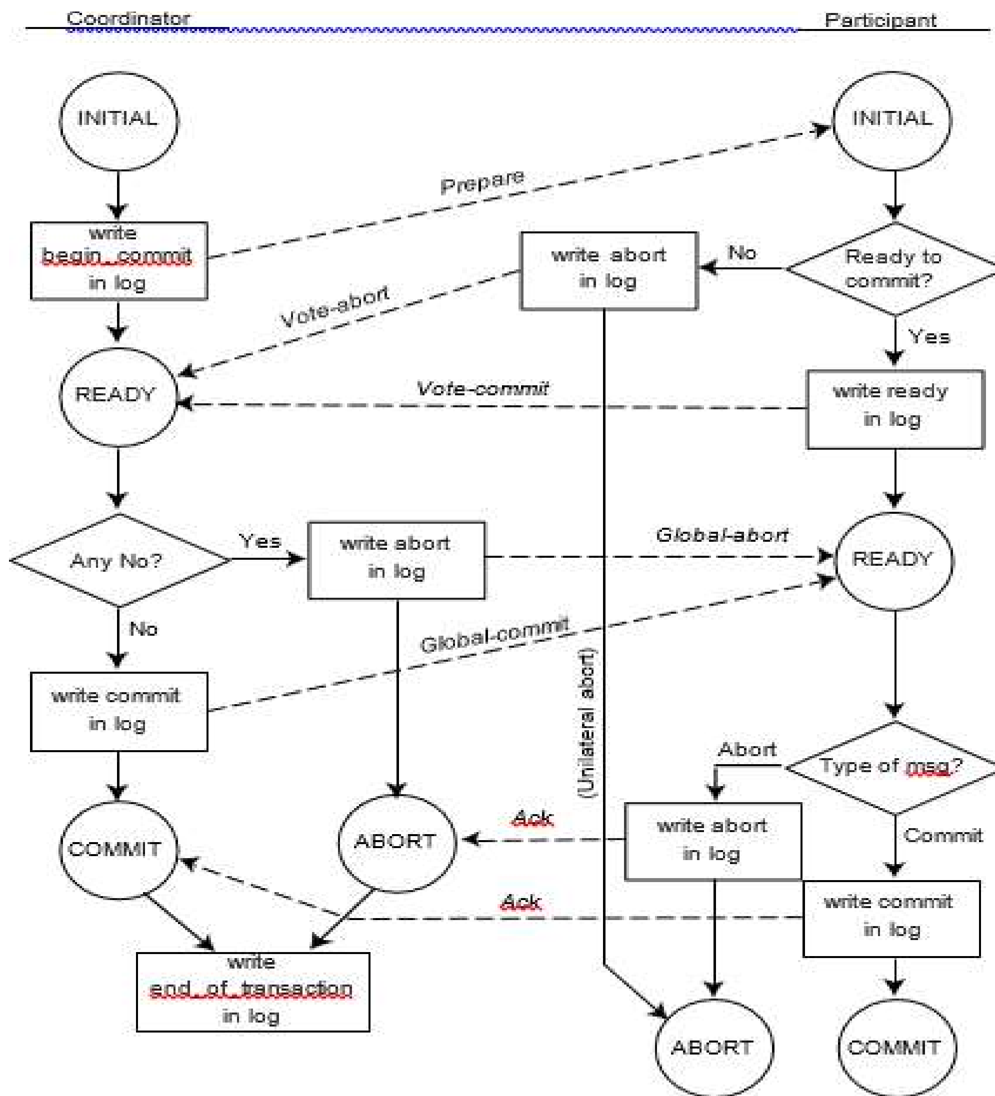

Fig.: 2PC Protocol Actions

Centralized vs. Linear/nested 2PC

- Centralized – participants do not communicate among themselves
- Linear – participants can communicate with one another
- Sites in linear system become numbered, pass messages to one another with "**vote-commit**" or "**vote-abort**" messages
- If last node decides to commit, send message back to coordinator node-by-node
- Distributed 2PC eliminates second phase of linear protocol
- Coordinator sends Prepare message to all participants
- Each participant sends its decision to all other participants and coordinator
- Participants must know identity of other participants for linear and distributed, not centralized

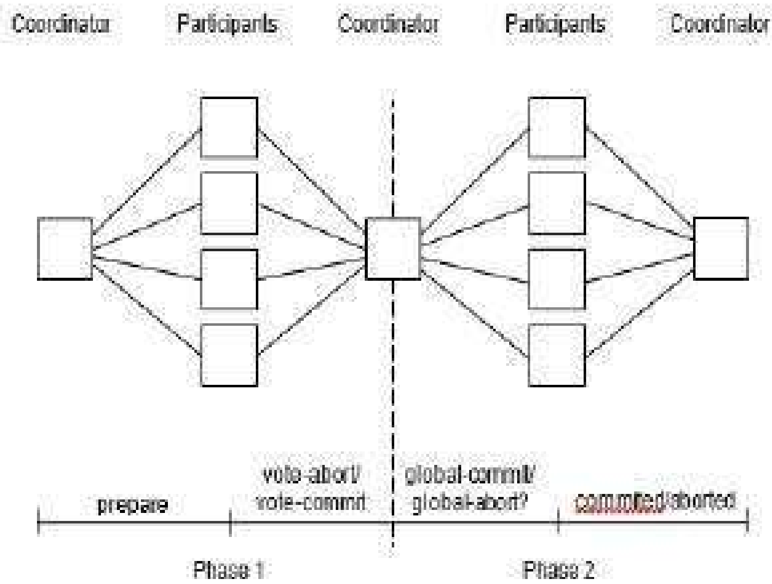

Fig. Centralized 2PC Communication Structure

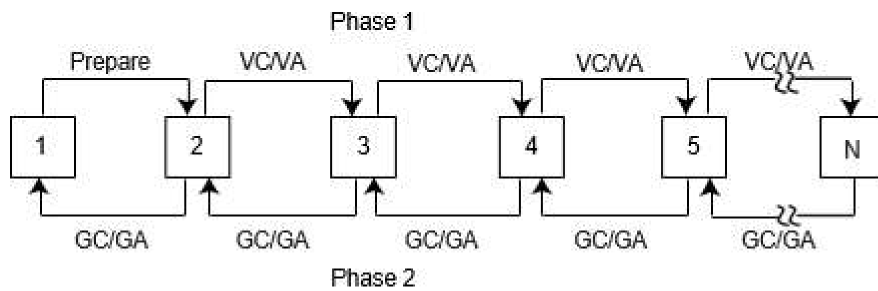

Fig. Linear 2PC Communication Structure. VC, vote.commit; VA, vote.abort; GC, global.commit; GA, global.abort.)

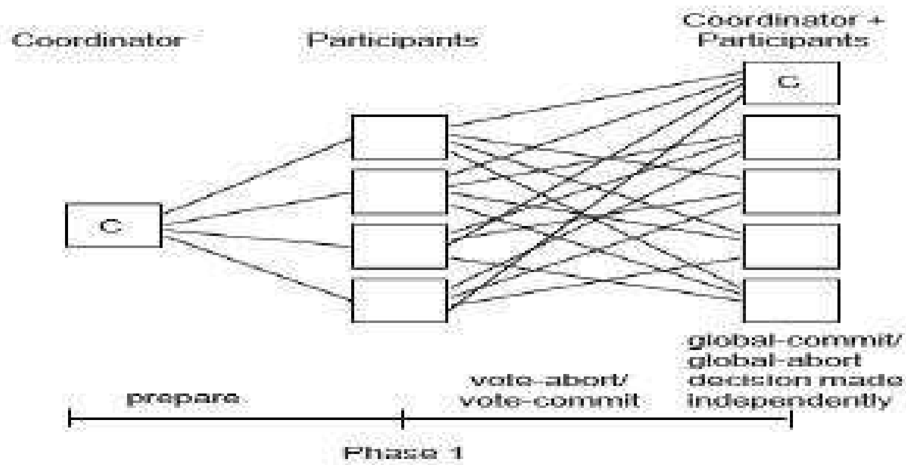

Fig. Distributed 2PC Communication Structure

Variations of 2PC

- Presumed Abort 2PC Protocol
  - Participant polls coordinator and there is no information about transaction, aborts
  - Coordinator can forget about transaction immediately after aborting
  - Expected to be more efficient, saves message transmission between coordinator and participants
- Presumed Commit 2PC Protocol
  - Likewise, if no information about transaction exists, should be considered committed
  - Most transactions are expected to commit
  - Could cause inconsistency – must create a collecting record

Site Failures and Network Partitioning

Dealing with Site Failures

- Let us first set the boundaries for the existence of non-blocking termination and independent recovery protocols in the presence of site failures.
- It can formally be proven that such protocols exist when a single site fails.
- In the case of multiple site failures, however, the prospects are not as promising.
- A negative result indicates that it is not possible to design independent recovery protocols (and, therefore, non- blocking termination protocols) when multiple sites fail.
- We first develop termination and recovery protocols for the 2PC algorithm and show that 2PC is inherently blocking.
- We then proceed to the development of atomic commit protocols which are non-blocking in the case of single site failures.

1. Termination and Recovery Protocols for 2PC

Termination Protocols

- The termination protocols serve the timeouts for both the coordinator and the participant processes.
- A timeout occurs at a destination site when it cannot get an expected message from a source site within the expected time period. In this section we consider that this is due to the failure of the source site.
- The method for handling timeouts depends on the timing of failures as well as on the types of failures. We therefore need to consider failures at various points of 2PC execution.
- State transition diagram :
- The states are denoted by circles and the edges represent the state transitions.
- The terminal states are depicted by concentric circles.
- The interpretation of the labels on the edges is as follows:

- the reason for the state transition, which is a received message, is given at the top, and the message that is sent as a result of state transition is given at the bottom.

Coordinator Timeouts.

- There are three states in which the coordinator can timeout: WAIT, COMMIT, and ABORT.
- Timeouts during the last two are handled in the same manner.
- So we need to consider only two cases:

1. *Timeout in the WAIT state*. In the WAIT state, the coordinator is waiting for the local decisions of the participants. The coordinator cannot unilaterally commit the transaction since the global commit rule has not been satisfied. However, it can decide to globally abort the transaction, in which case it writes an abort record in the log and sends a "global-abort" message to all the participants.
2. *Timeout in the COMMIT or ABORT states*. In this case the coordinator is not certain that the commit or abort procedures have been completed by the local recovery managers at all of the participant sites. Thus the coordinator repeatedly sends the "global-commit" or "global-abort" commands to the sites that have not yet responded, and waits for their acknowledgement.

Participant Timeouts.

- A participant can time out in two states: INITIAL and READY.
- Let us examine both of these cases.

1. *Timeout in the INITIAL state*. In this state the participant is waiting for a "prepare" message. The coordinator must have failed in the INITIAL state. The participant can unilaterally abort the transaction following a timeout. If the "prepare" message arrives at this participant at a later time, this can be handled in one of two possible ways. Either the participant would check its log, find the abort record, and respond with a "vote-abort," or it can simply ignore the "prepare" message. In the latter case the coordinator would time out in the WAIT state and follow the course we have discussed above.

2. *Timeout in the READY state*. In this state the participant has voted to commit the transaction but does not know the global decision of the coordinator. The participant cannot unilaterally make a decision. Since it is in the READY state, it must have voted to commit the transaction. Therefore, it cannot now change its vote and unilaterally abort it. On the other hand, it cannot unilaterally decide to commit it since it is possible that another participant may have voted to abort it. In this case the participant will remain blocked until it can learn from someone (either the coordinator or some other participant) the ultimate fate of the transaction.

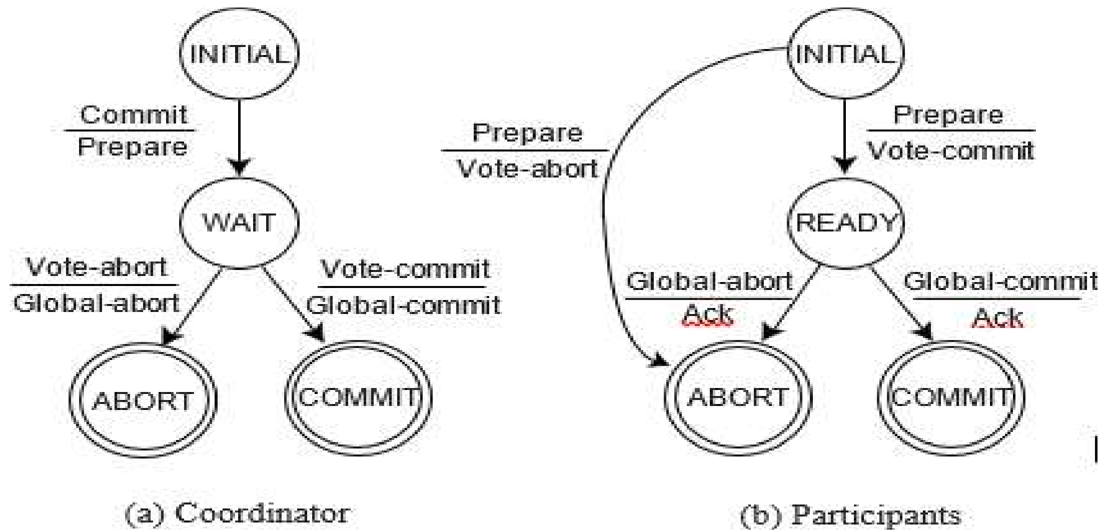

Fig. State Transitions in 2PC Protocol

Recovery Protocols

- In the preceding section, we discussed how the 2PC protocol deals with failures from the perspective of the operational sites.
- In this section, we take the opposite viewpoint: we are interested in investigating protocols that a coordinator or participant can use to recover their states when their sites fail and then restart.
- Remember that we would like these protocols to be independent.
- In general, it is not possible to design protocols that can guarantee independent recovery while

Algorithm 12.3: 2PC Coordinator Terminate

```
begin
    if in WAIT state then                          {coordinator is in ABORT state}
        write abort record in the log ;
        send "Global-abort" message to all the participants
    else                                           {coordinator is in COMMIT state}
        check for the last log record ;
        if last log record = abort then
            send "Global-abort" to all participants that have not responded
        else
            send "Global-commit" to all the participants that have not
            responded
    set timer ;
end
```

Algorithm 12.4: 2PC-Participant Terminate

```
begin
    if in INITIAL state then
        write abort record in the log
    else
        send "Vote-commit" message to the coordinator ;
        reset timer
end
```

maintaining the atomicity of distributed transactions.

- This is not surprising given the fact that the termination protocols for 2PC are inherently blocking.

Coordinator Site Failures

The following cases are possible:

1. *The coordinator fails while in the INITIAL state*. This is before the coordinator has initiated the commit procedure. Therefore, it will start the commit process upon recovery.
2. *The coordinator fails while in the WAIT state*. In this case, the coordinator has sent the “prepare” command. Upon recovery, the coordinator will restart the commit process for this transaction from the beginning by sending the “prepare” message one more time.
3. *The coordinator fails while in the COMMIT or ABORT states*. In this case, the coordinator will have informed the participants of its decision and terminated the transaction. Thus, upon recovery, it does not need to do anything if all the acknowledgments have been received. Otherwise, the termination protocol is involved.

Participant Site Failures

There are three alternatives to consider:

1. *A participant fails in the INITIAL state*. Upon recovery, the participant should abort the transaction unilaterally. Let us see why this is acceptable. Note that the coordinator will be in the INITIAL or WAIT state with respect to this transaction. If it is in the INITIAL state, it will send a “prepare” message and then move to the WAIT state. Because of the participant site’s failure, it will not receive the participant’s decision and will time out in that state. We have already discussed how the coordinator would handle timeouts in the WAIT state by globally aborting the transaction.
2. *A participant fails while in the READY state*. In this case the coordinator has been informed of the failed site’s affirmative decision about the transaction before the failure. Upon recovery, the participant at the failed site can treat this as a timeout in the READY state and hand the incomplete transaction over to its termination protocol.
3. *A participant fails while in the ABORT or COMMIT state*. These states represent the termination conditions, so, upon recovery, the participant does not need to take any special action.

2. Three-Phase Commit Protocol

- The three-phase commit protocol (3PC) is designed as a non-blocking protocol.
- It is indeed non-blocking when failures are restricted to site failures.
- Let us first consider the necessary and sufficient conditions for designing non- blocking atomic commitment protocols.
- A commit protocol that is synchronous within one state transition is non-blocking if and only if its state transition diagram contains neither of the following:

    1. No state that is “adjacent” to both a commit and an abort state.
    2. No non-committable state that is “adjacent” to a commit state.

- The term *adjacent* here means that it is possible to go from one state to the other with a single state transition.
  - Consider the COMMIT state in the 2PC protocol. If any process is in this state, we know that all the sites have voted to commit the transaction. Such states are called *committable*. There are other states in the 2PC protocol that are *non-committable*.
  - The one we are interested in is the READY state, which is non-committable since the existence of a process in this state does not imply that all the processes have voted to commit the transaction.
  - It is obvious that the WAIT state in the coordinator and the READY state in the participant 2PC protocol violate the non-blocking conditions we have stated above.
  - Therefore, one might be able to make the following modification to the 2PC protocol to satisfy the conditions and turn it into a non-blocking protocol.
  - We can add another state between the WAIT (and READY) and COMMIT states which serves as a buffer state where the process is ready to commit (if that is the final decision) but has not yet committed.
  - This is called the three-phase commit protocol (3PC) because there are three state transitions from the INITIAL state to a COMMIT state.
  - Observe that 3PC is also a protocol where all the states are synchronous within one state transition. Therefore, the foregoing conditions for non-blocking 2PC apply to 3PC.

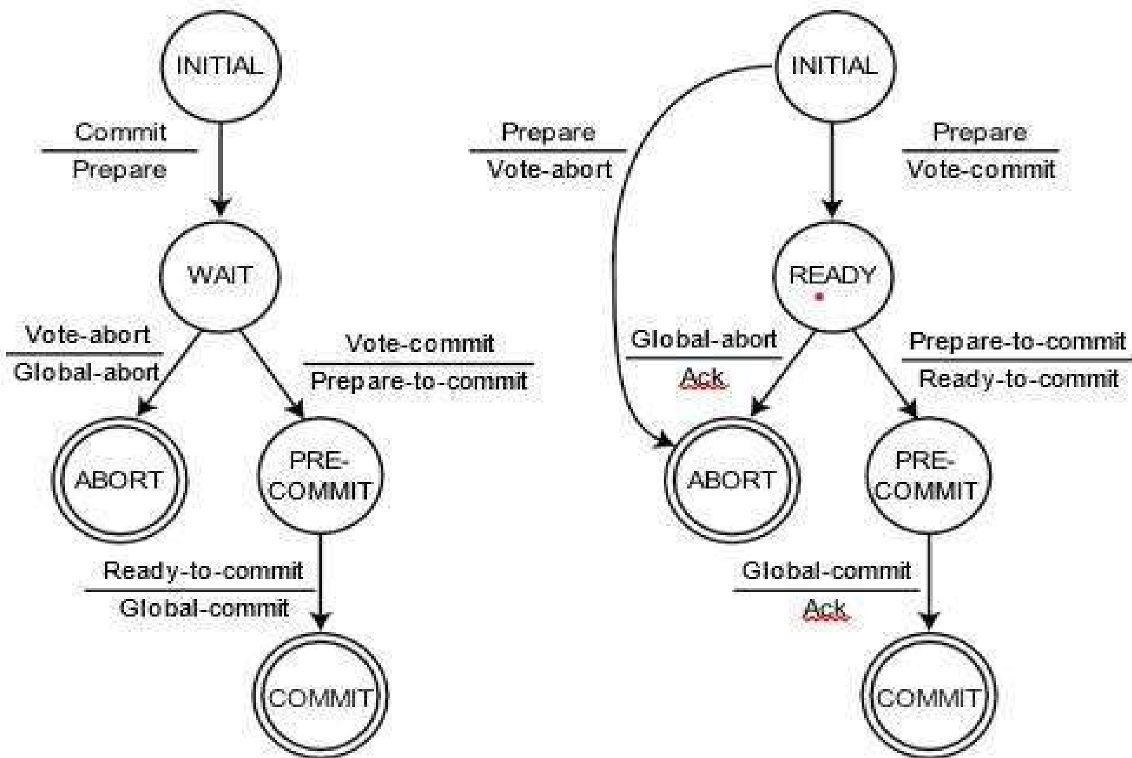

Fig. State Transitions in 3PC Protocol

- It is possible to design different 3PC algorithms depending on the communication topology.
- It is also straightforward to design a distributed 3PC protocol.

Termination Protocol

- As we did in discussing the termination protocols for handling timeouts in the 2PC protocol, let us investigate timeouts at each state of the 3PC protocol.

Coordinator Timeouts.

- In 3PC, there are four states in which the coordinator can time out: WAIT, PRECOM- MIT, COMMIT, or ABORT.

1. *Timeout in the WAIT state*. This is identical to the coordinator timeout in the WAIT state for the 2PC protocol. The coordinator unilaterally decides to abort the transaction. It therefore writes an abort record in the log and sends a "global-abort" message to all the participants that have voted to commit the transaction.
2. *Timeout in the PRECOMMIT state*. The coordinator does not know if the non-responding participants have already moved to the PRECOMMIT state. However, it knows that they are at least in the READY state, which means that they must have voted to commit the transaction. The coordinator can therefore move all participants to PRECOMMIT state by sending a "prepare-to-commit" message go ahead and globally commit the transaction by writing a commit record in the log and sending a "global-commit" message to all the operational participants.
3. *Timeout in the COMMIT (or ABORT) state*. The coordinator does not know whether the participants have actually performed the commit (abort) command. However, they are at least in the PRECOMMIT (READY) state (since the protocol is synchronous within one state transition) and can follow the termination protocol as described in case 2 or case 3 below. Thus the coordinator does not need to take any special action.

Participant Timeouts.

- A participant can time out in three states: INITIAL, READY, and PRECOMMIT. Let us examine all of these cases.

1. *Timeout in the INITIAL state*. This can be handled identically to the termina- tion protocol of 2PC.
2. *Timeout in the READY state*. In this state the participant has voted to commit the transaction but does not know the global decision of the coordinator. Since communication with the coordinator is lost, the termination protocol proceeds by electing a new coordinator, as discussed earlier. The new coordinator then terminates the transaction according to a termination protocol that we discuss below.

3. *Timeout in the PRECOMMIT state*. In this state the participant has received the "prepare-to-commit" message and is awaiting the final "global-commit" message from the coordinator. This case is handled identically to case 2 above.

Recovery Protocols

- There are some minor differences between the recovery protocols of 3PC and those of 2PC.
- We only indicate those differences.

1. *The coordinator fails while in the WAIT state*. This is the case we discussed at length in the earlier section on termination protocols. The participants have already terminated the transaction. Therefore, upon recovery, the coordinator has to ask around to determine the fate of the transaction.
2. *The coordinator fails while in the PRECOMMIT state*. Again, the termination protocol has guided the operational participants toward termination. Since it is now possible to move from the PRECOMMIT state to the ABORT state during this process, the coordinator has to ask around to determine the fate of the transaction.
3. *A participant fails while in the PRECOMMIT state*. It has to ask around to determine how the other participants have terminated the transaction.

- One property of the 3PC protocol becomes obvious from this discussion.
- When using the 3PC protocol, we are able to terminate transactions without blocking.
- However, we pay the price that fewer cases of independent recovery are possible.
- This also results in more messages being exchanged during recovery.

Network Partitioning

- Network partitions are due to communication line failures and may cause the loss of messages, depending on the implementation of the communication subnet.
- A partitioning is called a ***simple partitioning*** if the network is divided into only two components; otherwise, it is called ***multiple partitioning***.
- No **non-blocking** atomic commitment protocol exists that is resilient to multiple partitioning.
- Possible to design non-blocking atomic commit protocol resilient to simple partitioning.
- Concern is with termination of transactions that were active at time of partitioning.
- Strategies: permit all partitions to continue normal operations, accept possible inconsistency of database; or block operation in some partitions to maintain consistency.
- This decision problem is the premise of a classification of partition handling strategies.
- We can classify the strategies as *pessimistic* or *optimistic*
- **Pessimistic strategies** emphasize the consistency of the database, and would therefore not permit transactions to execute in a partition if there is no guarantee that the consistency of the database can be maintained.
- **Optimistic approaches**, on the other hand, emphasize the availability of the database even if this would cause inconsistencies.

- All the known termination protocols that deal with network partitioning in the case of non-replicated databases are pessimistic.
- Since the pessimistic approaches emphasize the maintenance of database consistency, the fundamental issue that we need to address is which of the partitions can continue normal operations.
- We consider two approaches.
    1. Centralized Protocols
    2. Voting-based Protocols

1. Centralized Protocols

- Centralized termination protocols are based on the centralized concurrency control algorithms.
- It makes sense to permit the operation of the partition that contains the central site, since it manages the lock tables.
- Primary site techniques are centralized with respect to each data item.
- In this case, more than one partition may be operational for different queries.
- For any given query, only the partition that contains the primary site of the data items that are in the write set of that transaction can execute that transaction.
- Both of these are simple approaches that would work well, but they are dependent on the concurrency control mechanism employed by the distributed database manager.
- Furthermore, they expect each site to be able to differentiate network partitioning from site failures properly.
- This is necessary since the participants in the execution of the commit protocol react differently to the different types of failures.

2. Voting-based Protocols

Voting as a technique for managing concurrent data accesses has been proposed by a number of researchers.
The fundamental idea is that a transaction is executed if a majority of the sites vote to execute it.
The idea of majority voting has been generalized to voting with *quorums*.
Quo- rum-based voting can be used as a replica control method, as well as a commit method to ensure transaction atomicity in the presence of network partitioning.
In the case of non-replicated databases, this involves the integration of the voting principle with commit protocols.
Every site in the system is assigned a vote V_i. Let us assume that the total number of votes in the system is V, and the abort and commit quorums are V_a and V_c, respectively.
Then the following rules must be obeyed in the implementation of the commit protocol:

1. $V_a + V_c > V$, where $0 \leq V_a, V_c \leq V$.
2. Before a transaction commits, it must obtain a commit quorum V_c.
3. Before a transaction aborts, it must obtain an abort quorum V_a.

- The first rule ensures that a transaction cannot be committed and aborted at the same time. The next two rules indicate the votes that a transaction has to obtain before it can terminate one way or the

other.

- The integration of these rules into the 3PC protocol requires a minor modification of the third phase.
- For the coordinator to move from the PRECOMMIT state to the COMMIT state, and to send the "global-commit" command, it is necessary for it to have obtained a commit quorum from the participants. This would satisfy rule 2.
- Note that we do not need to implement rule 3 explicitly. This is due to the fact that a transaction which is in the WAIT or READY state is willing to abort the transaction. Therefore, an abort quorum already exists.
- Let us now consider the termination of transactions in the presence of failures.
- When a network partitioning occurs, the sites in each partition elect a new coordinator, similar to the 3PC termination protocol in the case of site failures.
- There is a fundamental difference, however. It is not possible to make the transition from the WAIT or READY state to the ABORT state in one state transition, for a number of reasons.
- Depending on the responses, it terminates the transaction as follows:

1. If at least one participant is in the COMMIT state, the coordinator decides to commit the transaction and sends a "global-commit" message to all the participants.
2. If at least one participant is in the ABORT state, the coordinator decides to abort the transaction and sends a "global-abort" message to all the participants.
3. If a commit quorum is reached by the votes of participants in the PRECOM- MIT state, the coordinator decides to commit the transaction and sends a "global-commit" message to all the participants.
4. If an abort quorum is reached by the votes of participants in the PREABORT state, the coordinator decides to abort the transaction and sends a "global- abort" message to all the participants.
5. If case 3 does not hold but the sum of the votes of the participants in the PRECOMMIT and READY states are enough to form a commit quorum, the coordinator moves the participants to the PRECOMMIT state by sending a "prepare-to-commit" message. The coordinator then waits for case 3 to hold.
6. Similarly, if case 4 does not hold but the sum of the votes of the participants in the PREABORT and READY states are enough to form an abort quorum, the coordinator moves the participants to the PREABORT state by sending a "prepare-to-abort" message. The coordinator then waits for case 4 to hold.

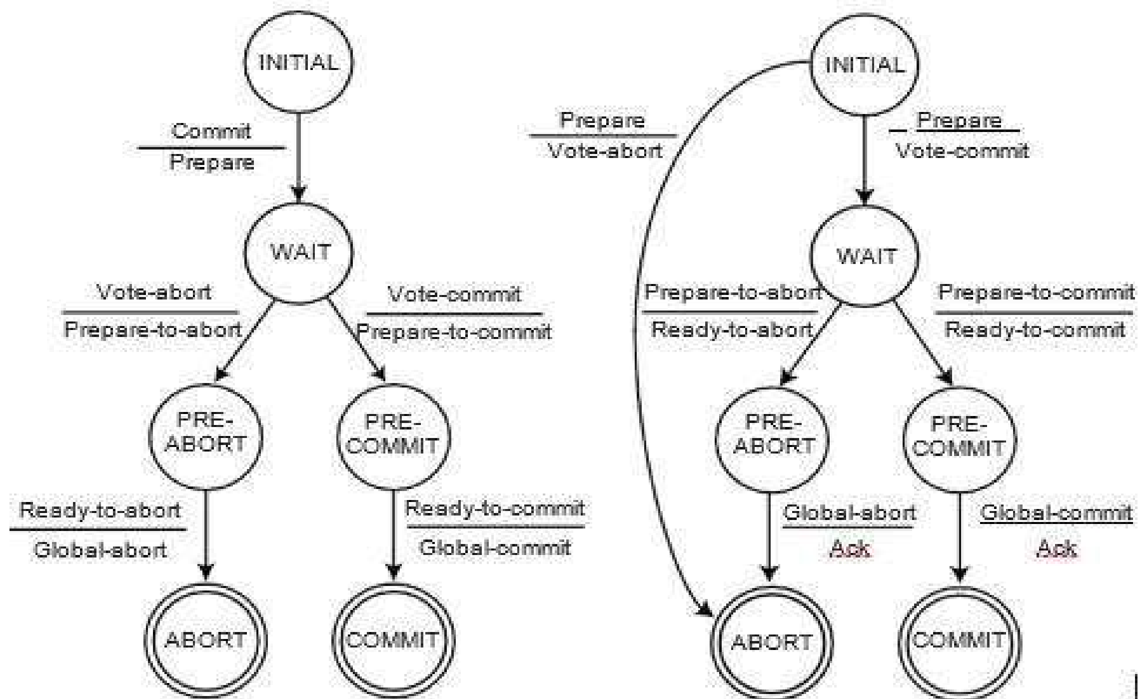

Fig. State Transitions in Quorum 3PC Protocol

- Two points are important about this quorum-based commit algorithm.
- First, it is blocking; the coordinator in a partition may not be able to form either an abort or a commit quorum if messages get lost or multiple partitioning occur. This is hardly surprising given the theoretical bounds that we discussed previously.
- The second point is that the algorithm is general enough to handle site failures as well as network partitioning. Therefore, this modified version of 3PC can provide more resiliency to failures.
- The recovery protocol that can be used in conjunction with the above-discussed termination protocol is very simple.
- When two or more partitions merge, the sites that are part of the new larger partition simply execute the termination protocol.
- That is, a coordinator is elected to collect votes from all the participants and try to terminate the transaction.

Parallel Database Systems

- Many data-intensive applications require support for very large databases (e.g., hundreds of terabytes or petabytes).
- Examples of such applications are e-commerce, data warehousing, and data mining.
- Very large databases are typically accessed through high numbers of concurrent transactions (e.g., performing on-line orders on an electronic store) or complex queries (e.g., decision-support queries). The first kind of access is representative of **On-Line Transaction Processing (OLTP)**

applications while the second is representative of **On-Line Analytical Processing (OLAP)** applications.

- Supporting very large databases efficiently for either OLTP or OLAP can be addressed by combining parallel computing and distributed database management.
- Data distribution can be exploited to increase performance (through parallelism) and availability (through replication).
- This principle can be used to implement ***parallel database systems***, i.e., database systems on parallel computers.
- Parallel database systems can exploit the parallelism in data management in order to deliver high-performance and high-availability database servers.
- Thus, they can support very large databases with very high loads.

Parallel Database System Architectures

- We present and compare the main architectures: shared-memory, shared-disk, shared-nothing and hybrid architectures.

Objectives

- A parallel database system can be loosely defined as a DBMS implemented on a parallel computer.
- The objectives of parallel database systems are covered by those of distributed DBMS (performance, availability, extensibility).
- A parallel database system should provide the following advantages.
    1. **High-performance**
    2. **High-availability.**
    3. **Extensibility**

High-performance.

- This can be obtained through several complementary solutions: database-oriented operating system support, parallel data management, query optimization, and load balancing
- Parallelism can increase throughput, using inter-query parallelism, and decrease transaction response times, using intra-query parallelism.
- Therefore, it is crucial to optimize and parallelize queries in order to minimize the overhead of parallelism, e.g., by constraining the degree of parallelism for the query.
- *Load balancing* is the ability of the system to divide a given workload equally among all processors.
- Depending on the parallel system architecture, it can be achieved statically by appropriate physical database design or dynamically at run-time.

High-availability.

- Because a parallel database system consists of many redundant components, it can well increase

data availability and fault-tolerance.

- A fault-tolerance technique that enables automatic redirection of transactions from a failed node to another node that stores a copy of the data.
- This provides uninterrupted service to users. However, it is essential that a node failure does not crate load imbalance, e.g., by doubling the load on the available copy.
- Solutions to this problem require partitioning copies in such a way that they can also be accessed in parallel.

Extensibility.

- In a parallel system, accommodating increasing database sizes or increasing performance demands (e.g., throughput) should be easier. Extensibility is the ability to expand the system smoothly by adding processing and storage power to the system.
- Ideally, the parallel database system should demonstrate two extensibility advantages : *linear speedup* and *linear scaleup.*
- Linear speedup refers to a linear increase in performance for a constant database size while the number of nodes (i.e., processing and storage power) are increased linearly.
- Linear scaleup refers to a sustained performance for a linear increase in both database size and number of nodes. Furthermore, extending the system should require minimal reorganization of the existing database.

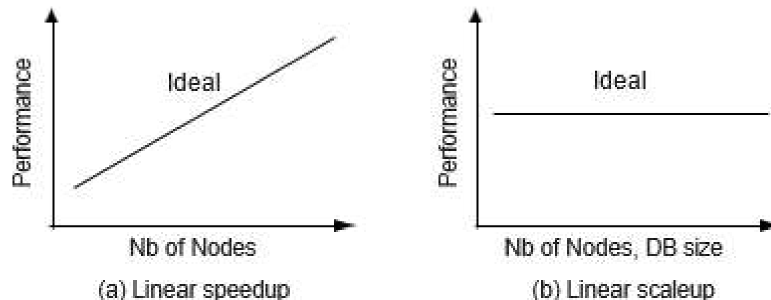

Fig. Extensibility Metrics

Functional Architecture

- Assuming a client/server architecture, the functions supported by a parallel database system can be divided into three subsystems much like in a typical DBMS.
- The differences, though, have to do with implementation of these functions, which must now deal with parallelism, data partitioning and replication, and distributed transactions.
- Depending on the architecture, a processor node can support all (or a subset) of these subsystems.

1. **Session Manager.**
    - It plays the role of a transaction monitor, providing support for client interactions with the server. In particular, it performs the connections and disconnections between the client processes and the two other subsystems.
    - Therefore, it initiates and closes user sessions (which may contain multiple transactions).

In case of OLTP sessions, the session manager is able to trigger the execution of pre-loaded transaction code within data manager modules.

2. **Transaction Manager.**
   - It receives client transactions related to query compilation and execution. It can access the database directory that holds all meta-information about data and programs.
   - The directory itself should be managed as a database in the server. Depending on the transaction, it activates the various compilation phases, triggers query execution, and returns the results as well as error codes to the client application.
   - Because it supervises transaction execution and commit, it may trigger the recovery procedure in case of transaction failure. To speed up query execution, it may optimize and parallelize the query at compile-time.
3. **Data Manager.**
   - It provides all the low-level functions needed to run compiled queries in parallel, i.e., database operator execution, parallel transaction support, cache management, etc.
   - If the transaction manager is able to compile dataflow control, then synchronization and communication among data man- ager modules is possible.
   - Otherwise, transaction control and synchronization must be done by a transaction manager module.

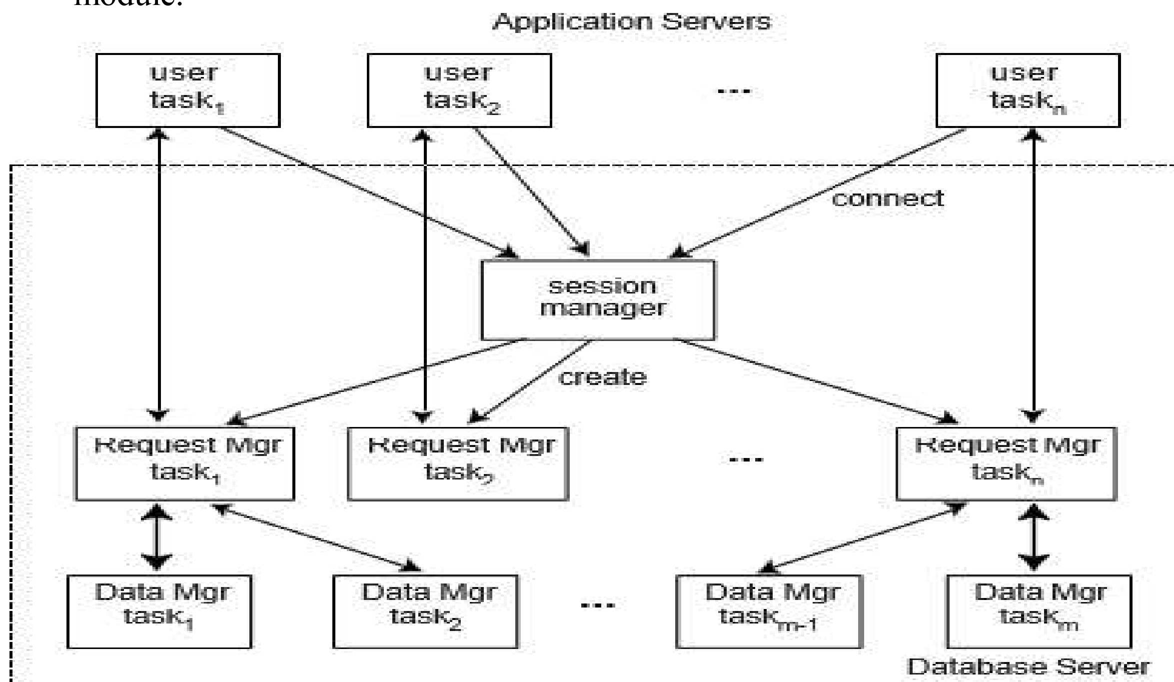

Fig. General Architecture of a Parallel Database System

Parallel DBMS Architectures

- There are three basic parallel computer architectures depending on how main memory or disk is

shared:

1. *shared-memory*
2. *shared-disk*
3. *shared-nothing*

- Hybrid architectures such as NUMA or *cluster* try to combine the benefits of the basic architectures.
- We focus on the four main hardware elements: interconnect, processors (P), main memory (M) and disks.
- We ignore other elements such as processor cache and I/O bus.

Shared-Memory

- In the shared-memory approach, any processor has access to any memory module or disk unit through a fast interconnect (e.g., a high-speed bus or a cross-bar switch).
- All the processors are under the control of a single operating system.
- All shared-memory parallel database products today can exploit inter-query parallelism to provide high transaction throughput and intra-query parallelism to reduce response time of decision-support queries.
- Current mainframe designs and symmetric multiprocessors (SMP) follow this approach.
- Since meta-information (directory) and control information (e.g., lock tables) can be shared by all processors.

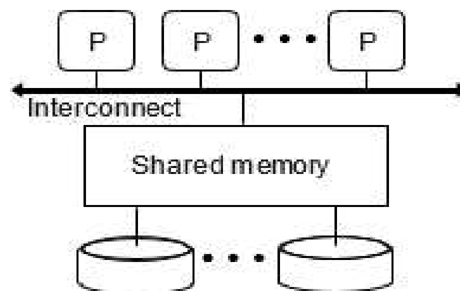

Fig. Shared-Memory Architecture

Shared-memory has two strong advantages: simplicity and load balancing.
Shared-memory has three problems: high cost, limited extensibility and low availability.

Shared-Disk

- In the shared-disk approach, any processor has access to any disk unit through the interconnect but exclusive (non-shared) access to its main memory.
- Each processor-memory node is under the control of its own copy of the operating system.
- Then, each processor can access database pages on the shared disk and cache them into its own memory.
- The first parallel DBMS that used shared-disk is Oracle with an efficient implementation of a distributed lock manager for cache consistency.

- Other major DBMS vendors such as IBM, Microsoft and Sybase provide shared-disk implementations.

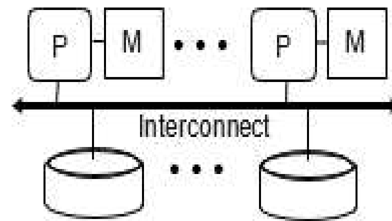

Fig. Shared-Disk Architecture

Shared-disk has a number of advantages: lower cost, high extensibility, load balancing, availability, and easy migration from centralized systems.
Shared-disk suffers from two problems: higher complexity and potential performance

Shared-Nothing

- In the shared-nothing approach each processor has exclusive access to its main memory and disk unit(s).
- Similar to shared-disk, each processor- memory-disk node is under the control of its own copy of the operating system.
- Then, each node can be viewed as a local site (with its own database and software) in a distributed database system.
- As opposed to SMP, this architecture is often called Massively Parallel Processor (MPP).

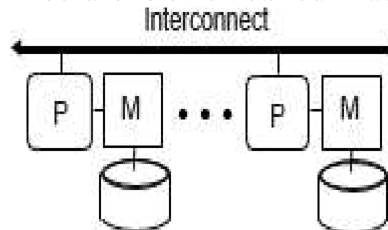

Fig. Shared-Nothing Architecture

Advantages : lower cost, high extensibility, and high availability.
Problems : Shared-nothing is much more complex to manage than either shared-memory or shared-disk.

Hybrid Architectures

- Hybrid architectures try to obtain the advantages of different architectures: typically the efficiency and simplicity of shared-memory and the extensibility and cost of either shared disk or shared nothing.
- There are two popular hybrid architectures: NUMA and cluster.

NUMA

- The objective of NUMA is to provide a shared-memory programming model and all its benefits, in a scalable architecture with distributed memory.
- The term NUMA reflects the fact that an access to the (virtually) shared memory may have a different cost depending on whether the physical memory is local or remote to the processor.
- The most successful class of NUMA multiprocessors is Cache Coherent NUMA (CC-NUMA).

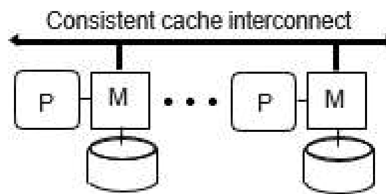

Fig. Cache coherent NUMA (CC-NUMA)

Cluster

- A cluster is a set of independent server nodes interconnected to share resources and form a single system. The shared resources, called *clustered* resources, can be hardware such as disk or software such as data management services.
- A cluster architecture has important advantages. It combines the flexibility and performance of shared-memory at each node with the extensibility and availability of shared-nothing or shared-disk.

Parallel Data Placement

- Data placement in a parallel database system exhibits similarities with data fragmentation in distributed databases (see Chapter 3). An obvious similarity is that fragmentation can be used to increase parallelism.
- we use the terms *partitioning* and *partition* instead of horizontal fragmentation and horizontal fragment, respectively
- To contrast with the alternative strategy, which consists of *clustering* a relation at a single node. The term *declustering* is sometimes used to mean partitioning.
- Vertical fragmentation can also be used to increase parallelism and load balancing much as in distributed databases.
- Data placement must be done so as to maximize system performance, which can be measured by combining the total amount of work done by the system and the response time of individual queries.
- An alternative solution to data placement is *full partitioning*, whereby each relation is horizontally fragmented across *all* the nodes in the system.
- There are three basic strategies for data partitioning:
    1. Round-Robin Partitioning

2. Hash Partitioning
3. Range Partitioning

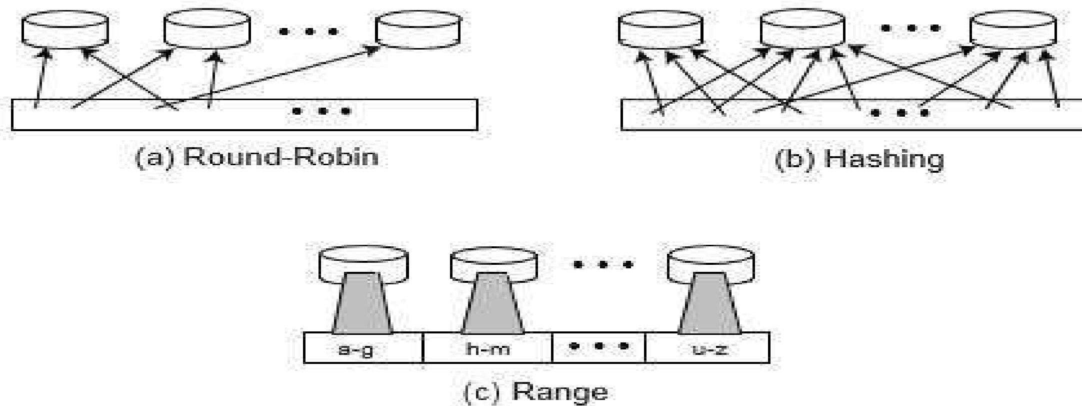

Fig. Different Partitioning Schemes

1. ***Round-robin partitioning***
   - *Round-robin partitioning* is the simplest strategy, it ensures uniform data distribution.
   - This strategy enables the sequential access to a relation to be done in parallel.
   - The direct access to individual tuples, based on a predicate, requires accessing the entire relation.
2. ***Hash partitioning***
   - *Hash partitioning* applies a hash function to some attribute that yields the partition number.
   - This strategy allows exact-match queries on the selection attribute to be processed by exactly one node and all other queries to be processed by all the nodes in parallel.
3. ***Range partitioning***
   - *Range partitioning* distributes tuples based on the value intervals (ranges) of some attribute.
   - In addition to supporting exact-match queries (as in hashing), it is well-suited for range queries.
   - For instance, a query with a predicate "A between A_1 and A_2" may be processed by the only node(s) containing tuples whose A value is in range [A_1, A_2]. However, range partitioning can result in high variation in partition size.

Parallel Query Processing

- The objective of parallel query processing is to transform queries into execution plans that can be efficiently executed in parallel.
- This is achieved by exploiting parallel data placement and the various forms of parallelism offered by high-level queries.

- The various forms of parallelism.
  - Query parallelism
  - Parallel algorithms for data processing
  - Parallel query optimization.

1. Query Parallelism

- Parallel query execution can exploit two forms of parallelism: inter- and intra-query *parallelism.*
- **Inter-query parallelism** which gives each machine different queries to work on so that the system can achieve a high throughput and complete as many queries as possible.
- **Intra-query parallelism** attempts to make one query run as fast as possible by spreading the work over multiple computers. We can further divide Intra-query parallelism into two classes: intra-operator and inter-operator.

Intra-operator Parallelism

- Intra-operator parallelism is based on the decomposition of one operator in a set of independent sub-operators, called *operator instances*.
- This decomposition is done using static and/or dynamic partitioning of relations.

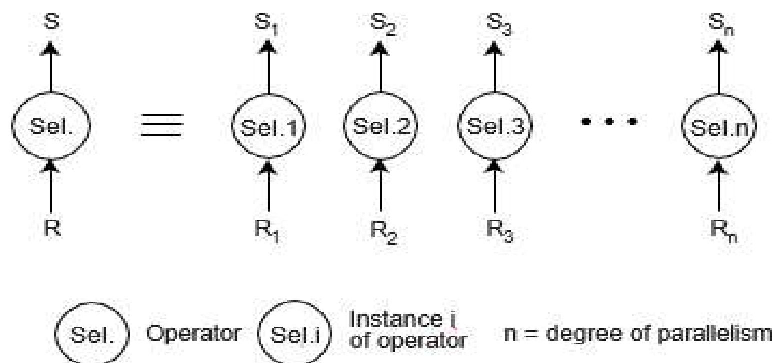

Fig. Intra-operator Parallelism

Inter-operator Parallelism

- Two forms of inter-operator parallelism can be exploited.
- *pipeline parallelism*, several operators with a producer-consumer link are executed in parallel. The select operator will be executed in parallel with the join operator.
- The advantage of such execution is that the intermediate result is not materialized, thus saving memory and disk accesses.
- *Independent parallelism* is achieved when there is no dependency between the operators that are executed in parallel. For instance, the two select operators can be executed in parallel.
- This form of parallelism is very attractive because there is no interference between the processors.

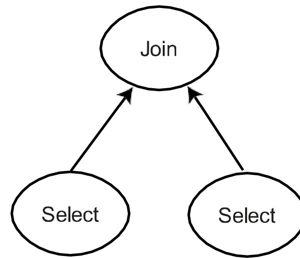

Fig. Inter-operator Parallelism

2. Parallel Algorithms for Data Processing

- Parallel data processing should exploit intra-operator parallelism. i.e, parallel algorithms for database operators on the select and join operators.
- The processing of the select operator in a partitioned data placement context is identical to that in a fragmented distributed database.
- The parallel processing of join is significantly more involved than that of select. The distributed join algorithms designed for high-speed networks can be applied successfully in a partitioned database context.
- There are three basic parallel join algorithms for partitioned databases:
  1. The Parallel Nested Loop (PNL) Algorithm
  2. The Parallel Associative Join (PAJ) Algorithm
  3. The Parallel Hash Join (PHJ) Algorithm.

- We describe each using a pseudo-concurrent programming language with three main constructs: **parallel- do, send**, and **receive**. **Parallel-do** specifies that the following block of actions is executed in parallel.
- For example,
  for i from 1 to *n* in parallel do action *A*
- Indicates that action *A* is to be executed by *n* nodes in parallel.
- **Send** and **receive** are the basic communication primitives to transfer data between nodes.

- To summarize, the parallel nested loop algorithm can be viewed as replacing the operator $R \; 0 \; S$ by $\bigcup_{i=1}^{n} (R \; 0 \; S_i)$.

Algorithm 14.1: PNL Algorithm

Input: $R_1, R_2, \ldots, R_m$: fragments of relation R;
$S_1, S_2, \ldots, S_n$: fragments of relation S;
JP: join predicate
Output: $T_1, T_2, \ldots, T_n$: result fragments
begin
  for *i from* 1 *to m in parallel* **do** {send R entirely to each S-node}
    send R_i to each node containing a fragment of S
  for *j from* 1 *to n in parallel* **do** {perform the join at each S-node}
    $R \leftarrow \bigcup_{i=1}^{m} R_i$; {receive R_i from R-nodes; R is fully replicated at S-nodes}
    $T_j \leftarrow R \; 0_{JP} \; S_j$
end

Example **: *T*he application of the parallel nested loop algorithm with *m* = *n* = 2.**

3. Parallel Query Optimization

- Parallel query optimization exhibits similarities with distributed query processing.
- It focuses much more on taking advantage of both intra-operator parallelism and inter-operator parallelism.
- A parallel query optimizer can be seen as three components: a search space, a cost model, and a search strategy.

Search Space

- Execution plans are abstracted by means of operator trees, which define the order in which the operators are executed.

Cost Model

- Cost model is responsible for estimating the cost of a given execution plan.
- It consists of two parts: architecture-dependent and architecture- independent.

Search Strategy

- The search strategy does not need to be different from either centralized or distributed query optimization.
- The search space tends to be much larger because there are more parameters that impact parallel execution plans, in particular, pipeline and store annotations.
- Thus, randomized search strategies generally outperform deterministic strategies in parallel query optimization.

Load Balancing

- Good load balancing is crucial for the performance of a parallel system.
- The response time of a set of parallel operators is that of the longest one.
- Minimizing the time of the longest one is important for minimizing response time.
- Balancing the load of different transactions and queries among different nodes is also essential to maximize throughput.
- Solutions to these problems can be obtained at the intra- and inter-operator levels

Intra-Operator Load Balancing

- Good intra-operator load balancing depends on the degree of parallelism and the allocation of processors for the operator.
- The skew problem makes it hard for a parallel query optimizer to make this decision statically (at compile-time) as it would require a very accurate and detailed cost model.
- Therefore, the main solutions rely on adaptive or specialized

Inter-Operator Load Balancing

- In order to obtain good load balancing at the inter-operator level, it is necessary to choose, for each operator, how many and which processors to assign for its execution.
- This should be done taking into account pipeline parallelism, which requires inter- operator communication.

Database Clusters

- Database clustering is the process of connecting more than one single database instance or server to your system.
- In most common database clusters, multiple database instances are usually managed by a single database server called the master.

Database Cluster Architecture

Database cluster is in two types of architectures

1. shared-disk architecture
2. shared-nothing architecture

Shared-Nothing Architecture

- To build a shared-nothing database architecture each database server must be independent of all other nodes.
- Meaning that each node has its own database server to store and access data from. In this type of

architecture, no single database server is master.

- Meaning that there is no one central database node that monitors and controls the access of data in the system.
- Note that a shared-nothing architecture offers great horizontal scalability as no resources are being shared between either nodes or database servers.

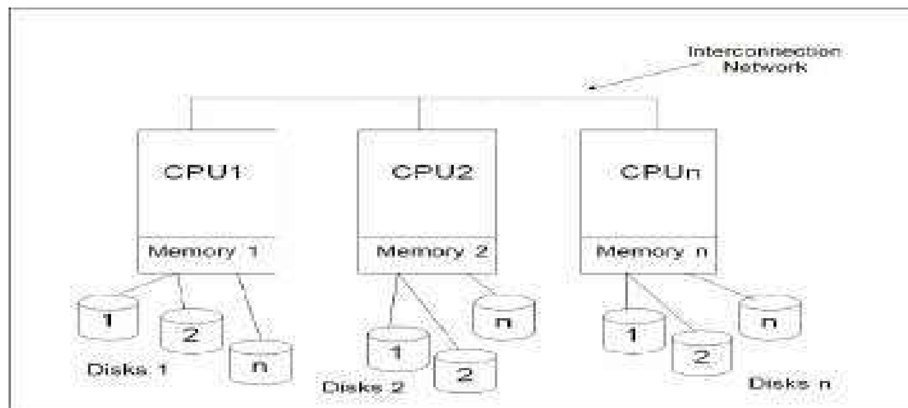

Fig : Shared-Nothing Architecture

Shared-Disk Architecture

- In this architecture, all nodes(CPU) share access to all the database servers available, subsequently having access to all the system's data.
- Unlike the shared-nothing architecture, the interconnection network layer is between the CPU and the database servers allowing for multiple database servers' access.
- It is worth noting that a shared disk cluster does not offer much scalability when compared to the shared-nothing architecture, as if all nodes share access to the same data a controlling node is required to monitor the data flow in the system.
- The issue is that after exceeding a certain number of slave nodes, the master node would be unable to monitor and control all the slave nodes efficiently.

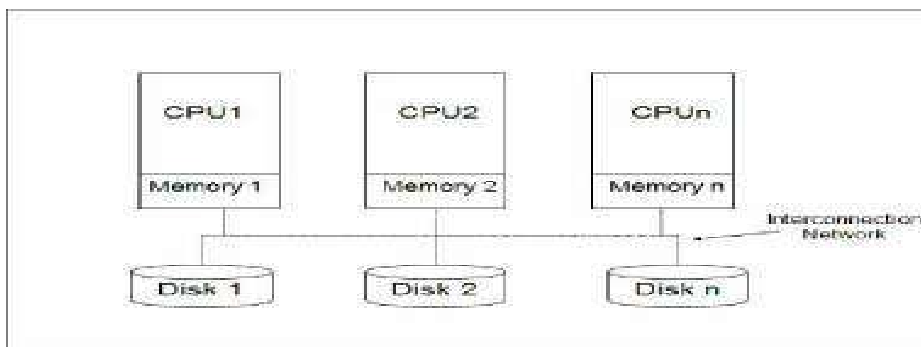

Fig : Shared Disk Architecture