Database Architecture

Database System Architectures

- Centralized Systems
- Client-Server Systems
- Parallel Systems
- Distributed Systems

(A)Centralized Systems

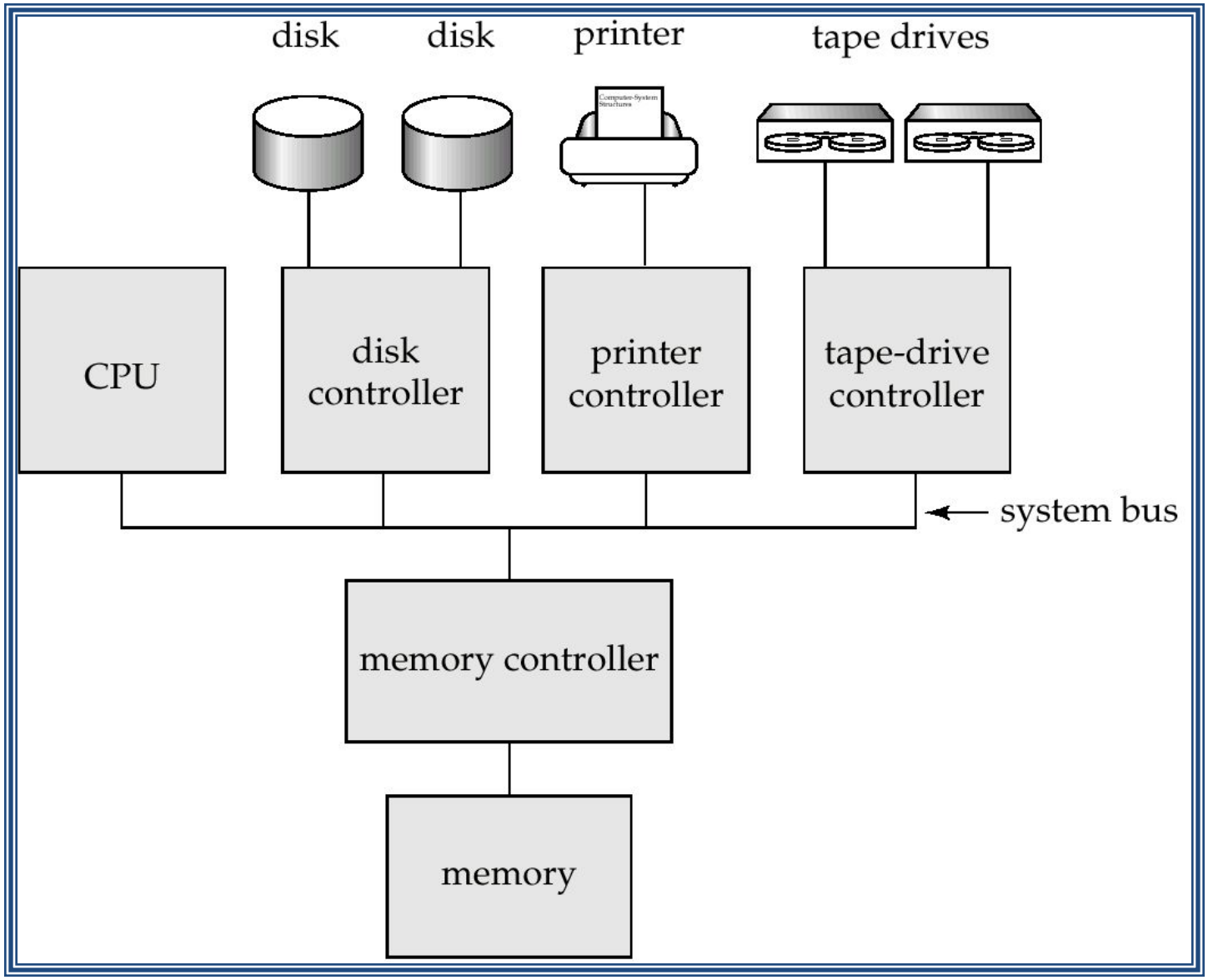
- Run on a single computer system and do not interact with other computer systems.

- General-purpose computer system: one to a few CPUs and a number of device controllers that are connected through a common bus that provides access to shared memory.

- **Single-user system** (e.g., personal computer or workstation): desk-top unit, single user, usually has only one CPU and one or two hard disks.

- **Multi-user system:** more disks, more memory, multiple CPUs. Serve a large number of users who are connected to the system via terminals. Often called *server* systems.

A Centralized Computer System



- Database systems designed for use by single users usually do not provide many of the facilities that a multiuser database provides.

- In particular, they may not support concurrency control, which is not required when only a single user can generate updates.

- Provisions for crash-recovery in such systems are either absent or primitive– for example, they may consist of simply making a backup of the database before any update.

- In contrast, database systems designed for multiuser systems support the full transactional features.

- Although general-purpose computer systems today have multiple processors, they have **coarse-granularity parallelism,** with only a few processors, all sharing the main memory.

- Databases running on such machines usually do not attempt to partition a single query among the processors; instead, they run each query on a single processor, allowing multiple queries to run concurrently.

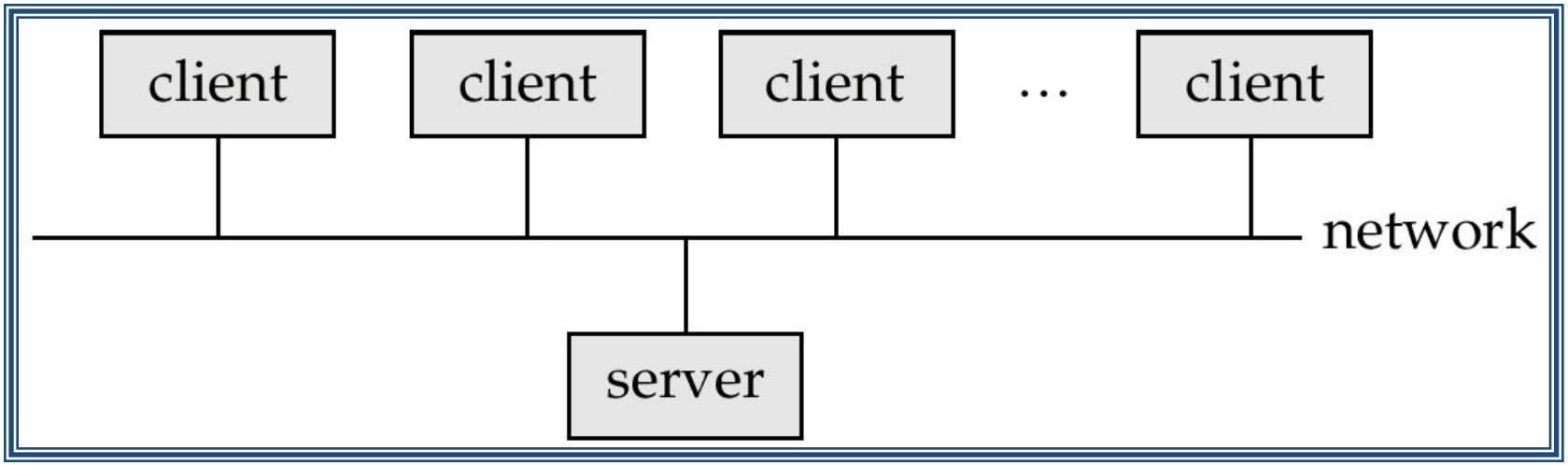
- Databases designed for single-processor machines already provide multitasking, allowing multiple processes to run on the same processor in a time-shared manner.

- Thus, coarse granularity parallel machines logically appear to be identical to single-processor machines, and database systems designed for time-shared machines can be easily adapted to run on them.

- In contrast, machines with **fine-granularity parallelism have a large number of** processors, and database systems running on such machines attempt to parallelize single tasks (queries, for example) submitted by users.

(B)Client-Server Systems

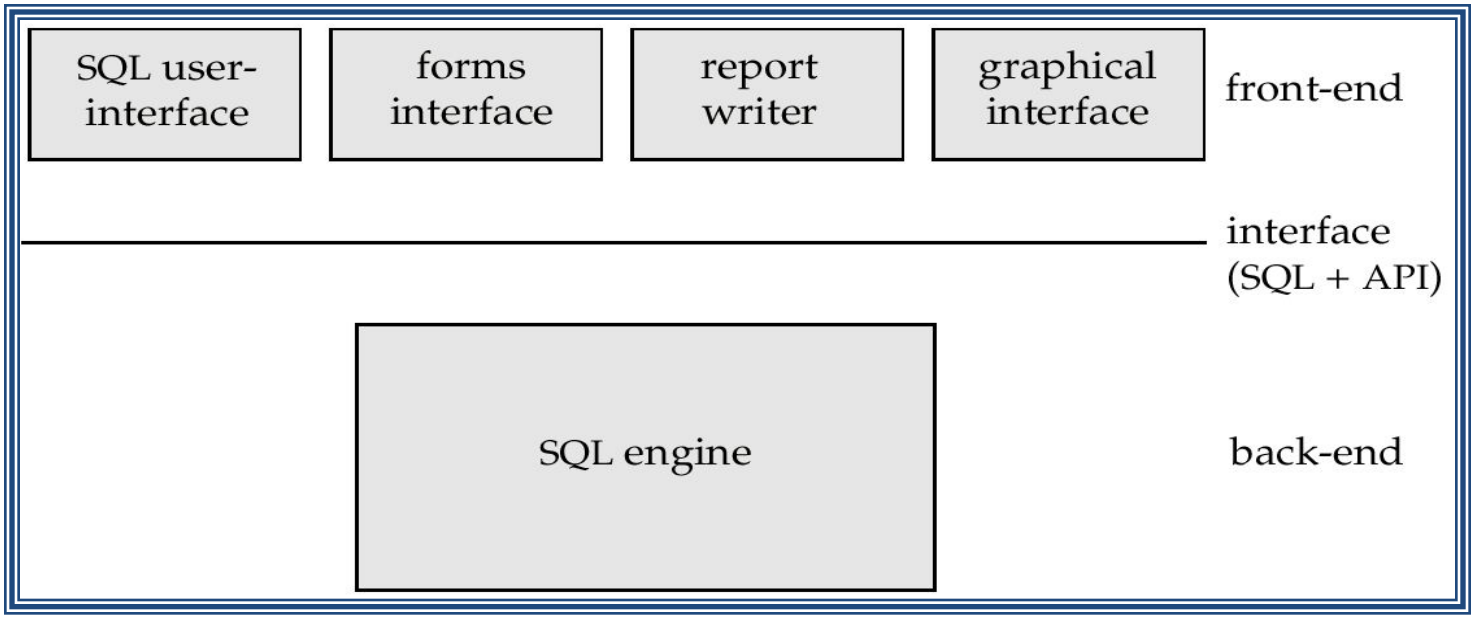
- Server systems satisfy requests generated at *m* client systems, whose general structure is shown below:



Continue……

- Database functionality can be divided into:
  - **Back-end**: manages access structures, query evaluation and optimization, concurrency control and recovery.
  - **Front-end**: consists of tools such as *forms*, *report-writers*, and graphical user interface facilities.
- The interface between the front-end and the back-end is through SQL or through an application program interface.

Front-end and back-end functionality



- Some transaction-processing systems provide a transactional remote procedure call interface to connect clients with a server.

- These calls appear like ordinary procedure calls to the programmer, but all the remote procedure calls from a client are enclosed in a single transaction at the server end.

- Thus, if the transaction aborts, the server can undo the effects of the individual remote procedure calls.

Server System Architectures

(a) Transaction Servers

- Also called **query server** systems or **SQL *server* systems**; clients send requests to the server system where the transactions are executed, and results are sent back to the client.
- Requests specified in SQL, and communicated to the server through a ***remote procedure call (RPC)*** mechanism.
- Transactional RPC allows many RPC calls to collectively form a transaction.
- *Open Database Connectivity (ODBC)* is a C language application program interface standard from Microsoft for connecting to a server, sending SQL requests, and receiving results.
- JDBC standard similar to ODBC, for Java

Transaction Server Process Structure

- A typical transaction server consists of multiple processes accessing data in shared memory.
- **(1) Server processes**
  - These are the processes that receive user queries (transactions), execute them and send results back.
  - The queries may be submitted to the server processes from a user interface, or from a user process running embedded SQL, or via JDBC, ODBC, or similar protocols.
  - Some database systems use a separate process for each user session, and a few use a single database process for all user sessions, but with multiple threads so that multiple queries can execute concurrently.

    i.e. Processes may be **multithreaded**, allowing a single process to execute several user queries concurrently.

- (A **thread** is like a process, but multiple threads execute as part of the same process, and all threads within a process run in the same virtual memory space. Multiple threads within a process can execute concurrently.)

- Many database systems use a hybrid architecture, with multiple processes, each one running multiple threads.

- **Lock manager process:** This process implements lock manager functionality, which includes lock grant, lock release, and deadlock detection.

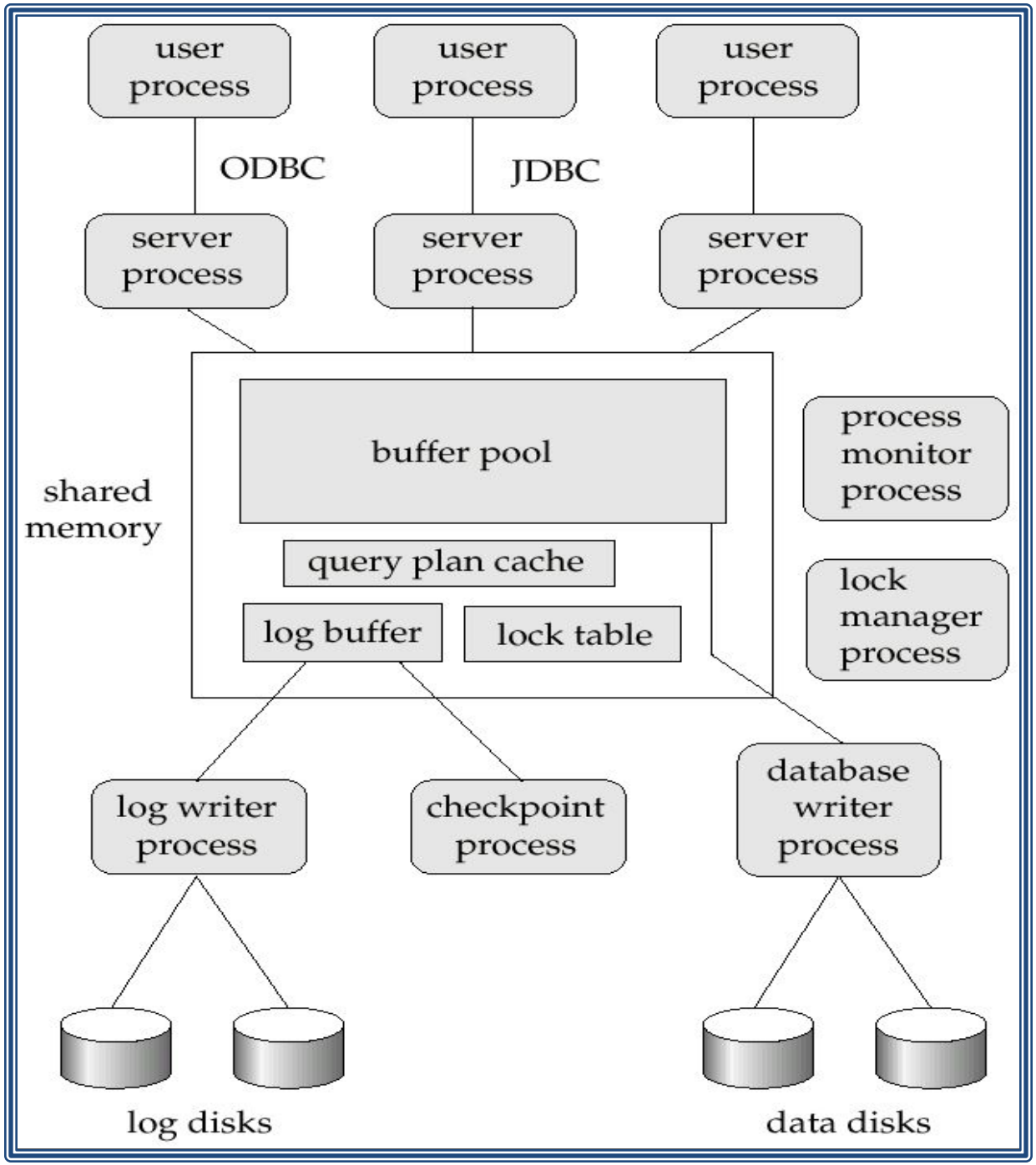
- **Database writer process:** There are one or more processes that output modified buffer blocks back to disk on a continuous basis.

- **Log writer process:** **This process outputs log records from the log record buffer** to stable storage. Server processes simply add log records to the log record buffer in shared memory, and if a log force is required, they request the log writer process to output log records.

- **Checkpoint process:** **This process performs periodic checkpoints.**

- **Process monitor process:** This process monitors other processes, and if any of them fails, it takes recovery actions for the process, such as aborting any transaction being executed by the failed process, and then restarting the process.

Shared memory and process structure



Transaction System Processes (Cont.)

- Shared memory contains shared data
  - Buffer pool
  - Lock table
  - Log buffer, containing log records waiting to be output to the log on stable storage
  - Cached query plans (reused if same query submitted again)
- All database processes can access shared memory.

- To ensure that no two processes are accessing the same data structure at the same time, databases systems implement **mutual exclusion** using operating system semaphores.

Continue…….

- Since multiple processes may read or perform updates on data structures in shared memory, there must be a mechanism to ensure that only one of them is modifying any data structure at a time, and no process is reading a data structure while it is being written by others.

- Such **mutual exclusion can be implemented by means of operating system functions** called semaphores.

- **The actions on lock request and release having two significant differences :**

- Since multiple server processes may access shared memory, mutual exclusion must be ensured on the lock table.
- If a lock cannot be obtained immediately because of a lock conflict, the lock request code keeps monitoring the lock table to check when the lock has been granted. The lock release code updates the lock table to note which process has been granted the lock.

- To avoid repeated checks on the lock table, operating system semaphores can be used by the lock request code to wait for a lock grant notification.
- The lock release code must then use the semaphore mechanism to notify waiting transactions that their locks have been granted.
- Even if the system handles lock requests through shared memory, it still uses the lock manager process for deadlock detection.

(b) Data Servers

- Data-server systems are used in local-area networks, where there is a high-speed connection between the clients and the server.

- In such an environment, it makes sense to ship data to client machines, to perform all processing at the client machine and then to ship the data back to the server machine.

- This architecture requires full back-end functionality at the clients.
- Issues:
  - Page-Shipping versus Item-Shipping
  - Locking
  - Data Caching
  - Lock Caching

 Page shipping versus item shipping. **The unit of communication for data can** be of coarse granularity, such as a page, or fine granularity, such as a tuple (or an object, in the context of object-oriented database systems).

- Here the term **item is used to refer to both tuples and objects.**
- If the unit of communication is a single item, the overhead of message passing is high compared to the amount of data transmitted.

- Instead, when an item is requested, it makes sense also to send back other items that are likely to be used in the near future.

- Fetching items even before they are requested is called **prefetching.**

- **Page shipping can be considered a form of prefetching if multiple** items reside on a page, since all the items in the page are shipped when a process desires to access a single item in the page.

Locking.

- Locks are usually granted by the server for the data items that it ships to the client machines.

- A disadvantage of page shipping is that a lock on a page implicitly locks all items contained in the page.

- Even if the client is not accessing some items in the page, it has implicitly acquired locks on all prefetched items.

- Other client machines that require locks on those items may be blocked unnecessarily.

- Techniques for lock **de-escalation, have been proposed where** the server can request its clients to transfer back locks on prefetched items.

- If the client machine does not need a prefetched item, it can transfer locks on the item back to the server, and the locks can then be allocated to other clients.

Data caching.

- Data that are shipped to a client on behalf of a transaction can be cached at the client, even after the transaction completes, if sufficient storage space is available.

- Successive transactions at the same client may be able to make use of the cached data.

- Even if a transaction finds cached data, it must make sure that those data are up to date, since they may have been updated by a different client after they were cached.

- Thus, a message must still be exchanged with the server to check validity of the data, and to acquire a lock on the data.

Lock caching.

- If the use of data is mostly partitioned among the clients, with clients rarely requesting data that are also requested by other clients, locks can also be cached at the client machine.

- Suppose that a client finds a data item in the cache, and that it also finds the lock required for an access to the data item in the cache.

- Then, the access can proceed without any communication with the server.

- However, the server must keep track of cached locks; if a client requests a lock from the server, the server must **call back all conflicting locks on** the data item from any other client machines that have cached the locks.

- The task becomes more complicated when machine failures are takes place.

(C)Parallel Systems

- Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network.
- A **coarse-grain parallel** machine consists of a small number of powerful processors
- A **massively parallel** or **fine grain parallel** machine utilizes thousands of smaller processors.
- Two main performance measures:
  - **throughput** --- the number of tasks that can be completed in a given time interval
  - **response time** --- the amount of time it takes to complete a single task from the time it is submitted.
- Two important issues in parallelism are **speedup and scaleup**.
- Running a given task in less time by increasing the degree of parallelism is called **speedup.**
- Handling larger tasks by increasing the degree of parallelism is called **scaleup.**

Speed-Up and Scale-Up

- **Speedup**: a fixed-sized problem executing on a small system is given to a system which is *N*-times larger.i.e. **Running a given task in less time by increasing the degree of parallelism.**

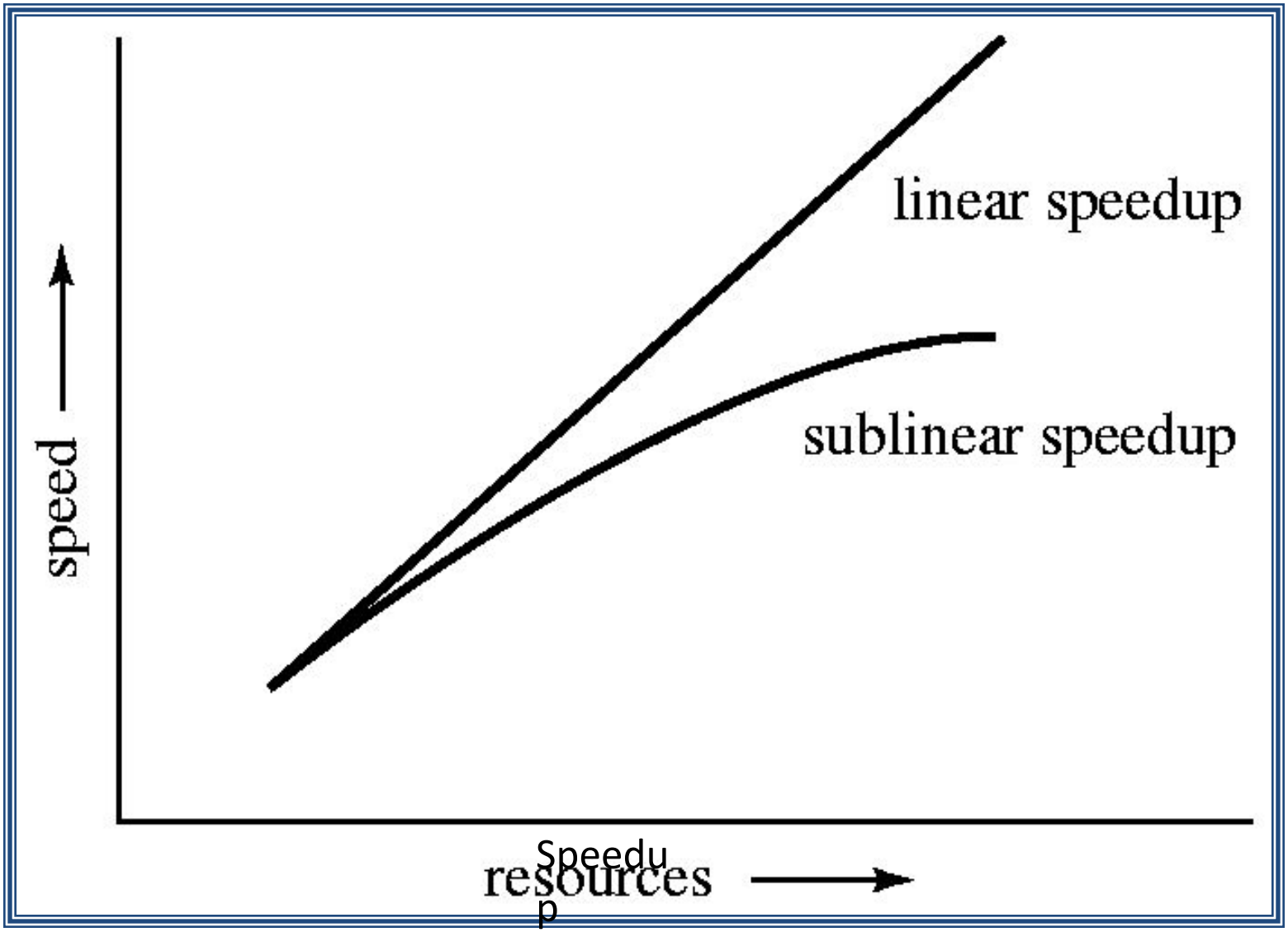
- Consider **the execution time of a task on the larger machine is TL** and the execution time of a same task on the smaller machine is **TS.**

- The speed up is measured due to parallelism as TS/TL.

- The parallel system is said to demonstrate **linear speedup** if the speedup is N. Where N is the number of resources used in larger systems.

- If the speedup is less than N,the system is said to demonstrate **sublinear speedup.**

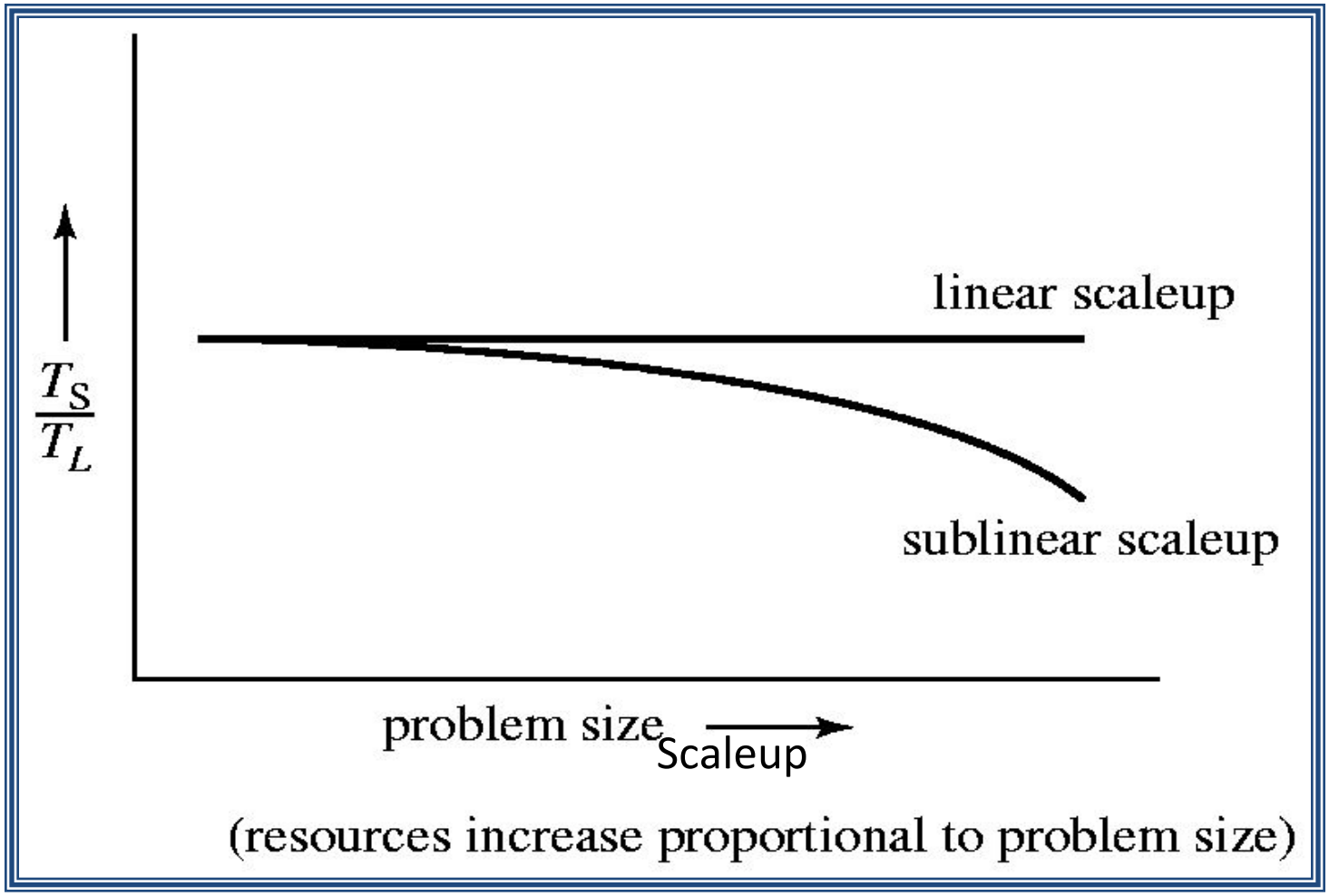
Speedup with increasing resources



❖**Scaleup**: increase the size of both the problem and the system
N-times larger system used to perform *N*-times larger job.**i.e.it relates to the ability to process larger tasks in the same amount of time by increasing more resources.**

- Suppose Q be a task and Qn is the task i.e. N times bigger than Q.
- Execution time of Q on machine is TS while execution time of Qn on parallel machine is TL, then the scaleup is calculated as TS/TL
- The parallel system is said to demonstrate **linear scaleup** on task Q if TL=TS.
- If TL>TS the system is said to demonstrate **sublinear scaleup.**

Scaleup with increasing problem size and resources



There are two kinds of **scaleup** that are relevant in parallel database systems, depending on how the size of the task is measured:

- In **batch scaleup, the size of the database increases, and the tasks are large jobs** whose runtime depends on the size of the database.

- An example of such a task is a scan of a relation whose size is proportional to the size of the database.

- Thus, the size of the database is the measure of the size of the problem.

- In **transaction scaleup, the rate at which transactions are submitted to the** database increases and the size of the database increases proportionally to the transaction rate.

- This kind of scaleup is what is relevant in transaction processing systems where the transactions are small updates—for example, a deposit or withdrawal from an account—and transaction rates grow as more accounts are created.

- Such transaction processing is especially well adapted for parallel execution, since transactions can run concurrently and independently on separate processors, and each transaction takes roughly the same amount of time, even if the database grows.

Factors Limiting Speedup and Scaleup

Speedup and scaleup are often sublinear due to:

- **(A) Startup costs**:

  -Cost of starting up multiple processes may dominate computation time, if the degree of parallelism is high.i.e.

  -There is a startup cost associated with initiating a single process.In a parallel operation consisting of thousands of processes, the *startup time* may overshadow the actual processing time.

- **(B) Interference**: Processes accessing shared resources (e.g.,system bus, disks, or locks) compete with each other, thus spending time waiting on other processes, rather than performing useful work.

(C)Skew:

- By breaking down a single task into a number of parallel steps, we reduce the size of the average step.

- It is often difficult to divide a task into exactly equal-sized parts, and the way that the sizes are distributed is therefore *skewed*.

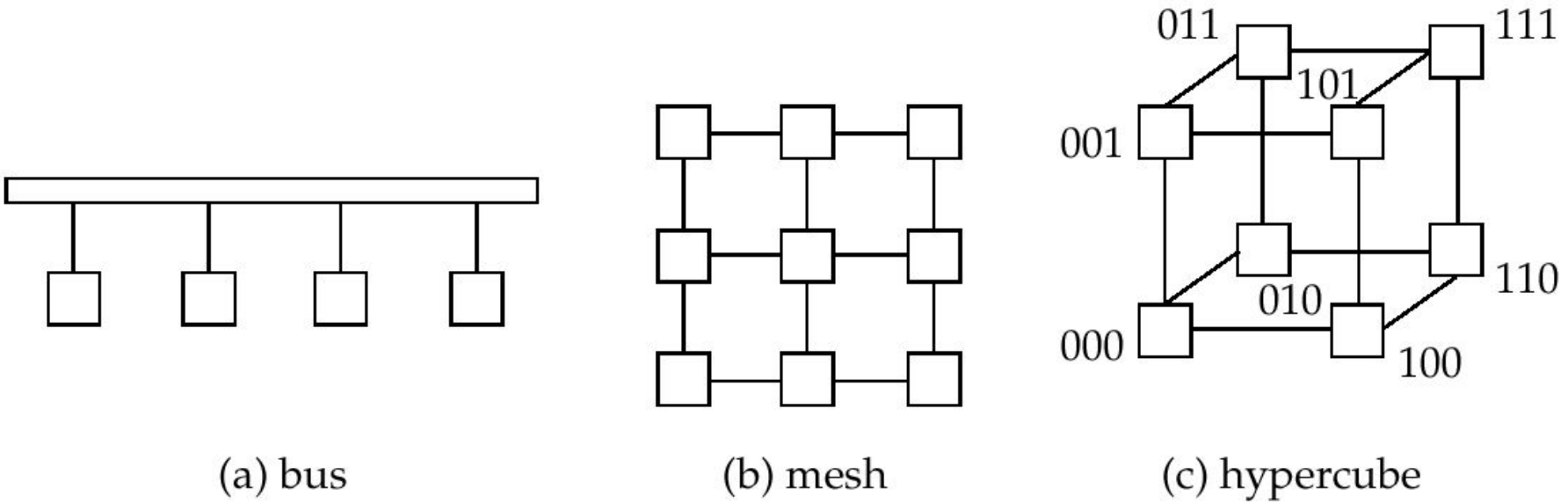
- Increasing the degree of parallelism increases the variance in service times of parallely executing tasks

- *For example, if a task of size 100 is* divided into 10 parts, and the division is skewed, there may be some tasks of size less than 10 and some tasks of size more than 10; if even one task happens to be of size 20, the speedup obtained by running the tasks in parallel is only Five.

Interconnection Network Architectures

- **Bus**. System components send data on and receive data from a single communication bus;
  - Does not scale well with increasing parallelism.
- **Mesh**. Components are arranged as nodes in a grid, and each component is connected to all adjacent components
  - Communication links grow with growing number of components, and so scales better.
- **Hypercube**. Components are numbered in binary; components are connected to one another if their binary representations differ in exactly one bit.
  - *n* components are connected to *log(n)* other components and can reach each other via at most *log(n)* links; reduces communication delays.

Interconnection Architectures



(a) bus (b) mesh (c) hypercube

Parallel Database Architectures

Parallel Database Architectures

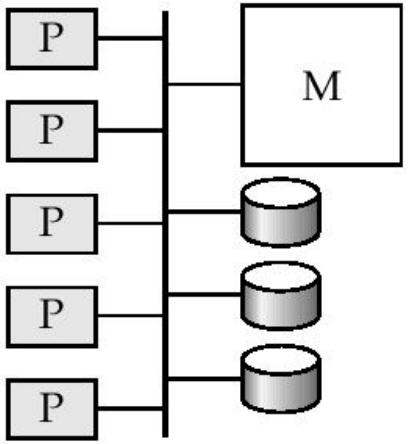
- **Shared memory** – All the processors share a common memory.

- **Shared disk** – All the processors share a common set of disks. Shared-disk systems are sometimes called **clusters.**

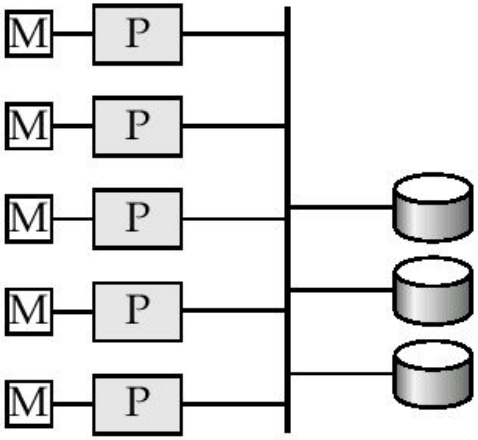
- **Shared nothing** – The processors share neither a common memory nor common disk.

- **Hierarchical** –This model is hybrid of the above architectures.

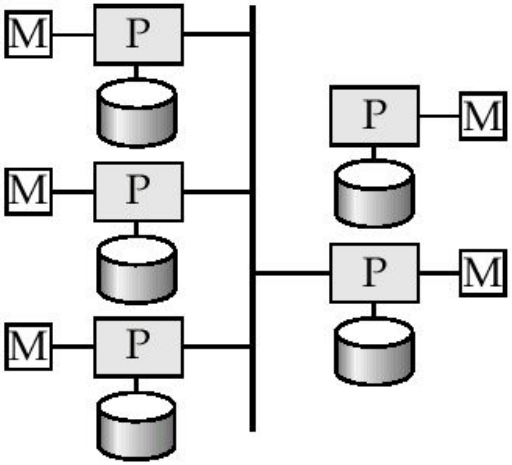
Parallel Database Architectures



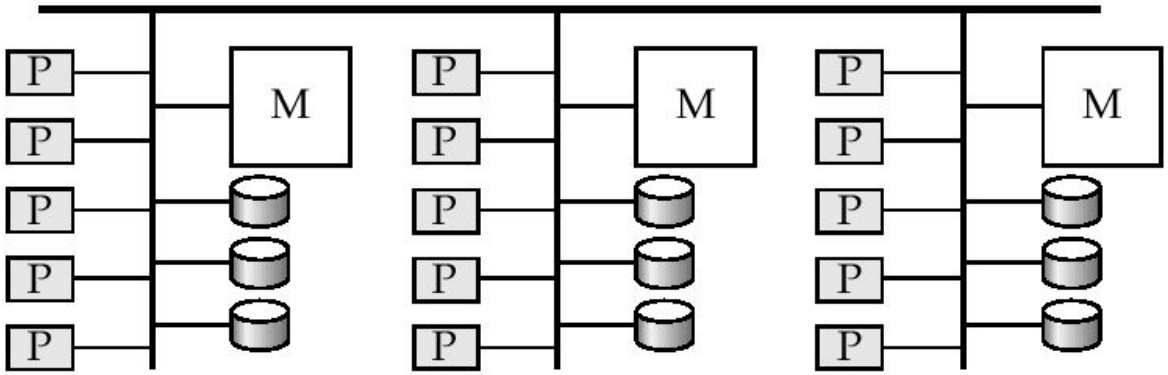
(a) shared memory



(b) shared disk



(c) shared nothing



(d) hierarchical

(I)Shared Memory

- In a **shared-memory architecture, the processors and disks have access to a common** memory, typically via a bus or through an interconnection network.
- The benefit of shared memory is extremely efficient communication between processors—data in shared memory can be accessed by any processor .
- A processor can send messages to other processors much faster by using memory writes than by sending a message through a communication mechanism.
- The downside of shared-memory machines is that the architecture is not scalable beyond 32 or 64 processors because the bus or the interconnection network becomes a bottleneck (since it is shared by all processors).
- Adding more processors does not help after a point, since the processors will spend most of their time waiting for their turn on the bus to access memory.

- Shared-memory architectures usually have large memory caches at each processor, so that referencing of the shared memory is avoided whenever possible.

- However, at least some of the data will not be in the cache, and accesses will have to go to the shared memory.

- Moreover, the caches need to be kept coherent; that is, if a processor performs a write to a memory location, the data in that memory location should be either updated at or removed from any processor where the data is cached.

- Maintaining cache-coherency becomes an increasing overhead with increasing number of processors.

(II)Shared Disk

- All processors can directly access all disks via an interconnection network, but the processors have private memories.

- There are two advantages of this architecture over a shared-memory architecture.

- First, since each processor has its own memory, the memory bus is not a bottleneck. Second, it offers a cheap way to provide a degree of **fault tolerance: If a processor (or its memory) fails,** the other processors can take over its tasks, since the database is resident on disks that are accessible from all processors.

- The main problem with a shared-disk system is again scalability.

- Although the memory bus is no longer a bottleneck, the interconnection to the disk subsystem is now a bottleneck; it is particularly so in a situation where the database makes a large number of accesses to disks.

- Compared to shared-memory systems, shared-disk systems can scale to a somewhat larger number of processors, but communication across processors is slower since it has to go through a communication network.

(C)Shared Nothing

- In a **shared-nothing system,** each node of the machine consists of a processor, memory, and one or more disks.

- The processors at one node may communicate with another processor at another node by a high-speed interconnection network.

- A node functions as the server for the data on the disk or disks that the node owns.

- Since local disk references are serviced by local disks at each processor, the shared-nothing model overcomes the disadvantage of requiring all I/O to go through a single interconnection network; only queries, accesses to nonlocal disks, and result relations pass through the network.

- Therefore, shared-nothing architectures are more scalable and can easily support a large number of processors.

- The main drawbacks of shared-nothing systems are the costs of communication and of nonlocal disk access, which are higher than in a shared-memory or shared-disk architecture since sending data involves software interaction at both ends.

- The Teradata database machine was among the earliest commercial systems to use the shared-nothing database architecture.

- The Grace and the Gamma research prototypes also used shared-nothing architectures.

(D)Hierarchical

- Combines characteristics of shared-memory, shared-disk, and shared-nothing architectures.

- Top level is a shared-nothing architecture – nodes connected by an interconnection network, and do not share disks or memory with each other.

- Each node of the system could be a shared-memory system with a few processors.

- Alternatively, each node could be a shared-disk system, and each of the systems sharing a set of disks could be a shared-memory system.

- Thus, a system could be built as a hierarchy, with shared-memory architecture with a few processors at the base, and a shared nothing architecture at the top, with possibly a shared-disk architecture in the middle.

- Commercial parallel database systems today run on several of these architectures.

Distributed database System

- In a **distributed database system,** the database is stored on several computers.

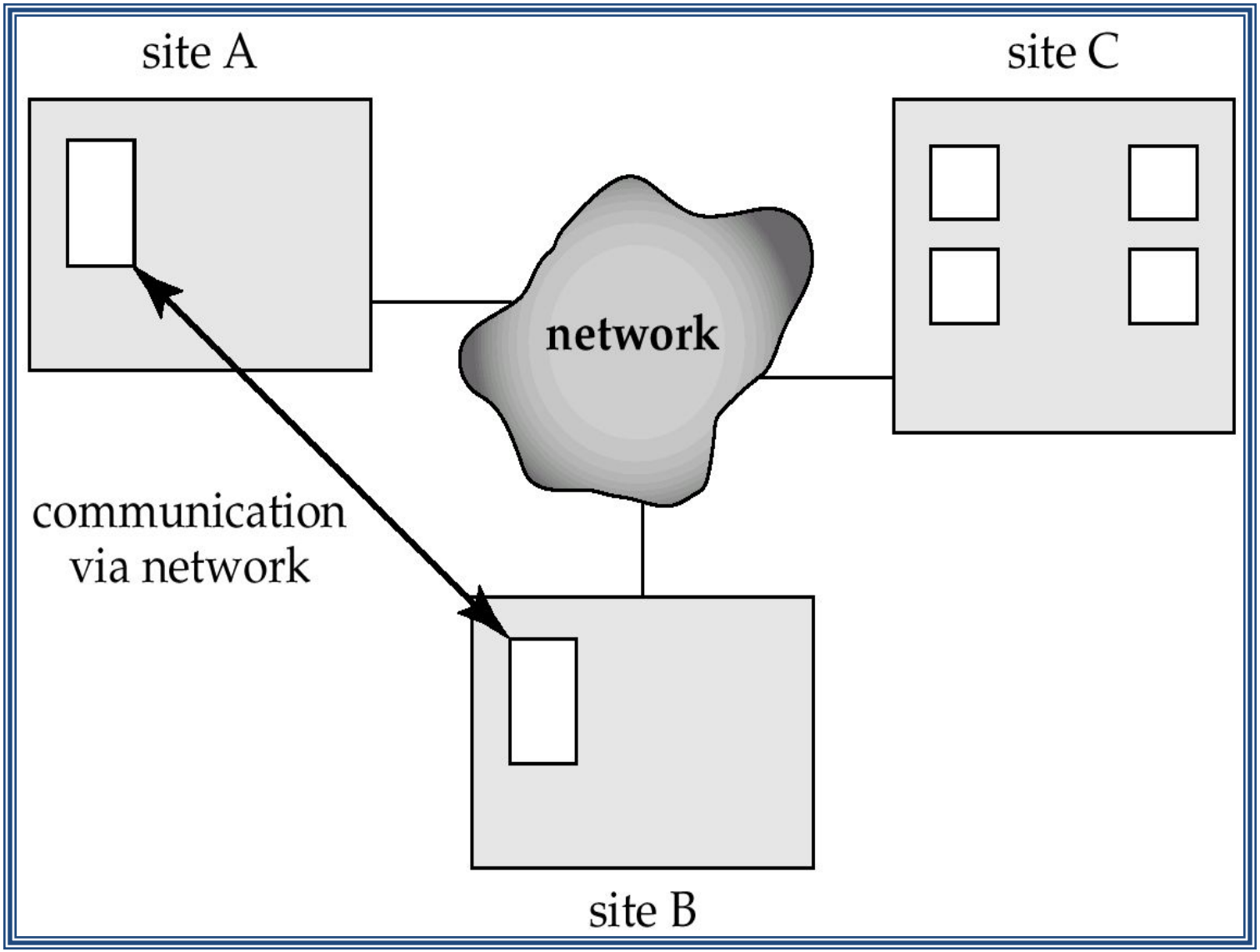
- The computers in a distributed system communicate with one another through various communication media, such as high-speed networks or telephone lines.

- They do not share main memory or disks.

- The computers in a distributed system may vary in size and function, ranging from workstations up to mainframe systems.

- The computers in a distributed system are referred to by a number of different names, such as **sites or nodes.**
- Here the term **site** is used , to highlight the physical distribution of these systems.
- The general structure of a distributed system is shown in Figure

A distributed system.



- The main differences between shared-nothing parallel databases and distributed databases are that distributed databases are typically geographically separated, are separately administered, and have a slower interconnection.

- Another major difference is that, in a distributed database system, we differentiate between local and global transactions.

- A **local transaction is one that accesses data only from sites** where the transaction was initiated.

- A **global transaction, on the other hand, is one** that either accesses data in a site different from the one at which the transaction was initiated, or accesses data in several different sites.

There are several reasons for building distributed database systems, including sharing of data, autonomy, and availability.

✔ ***Sharing data:***

- ***The major advantage in building a distributed database system*** is the provision of an environment where users at one site may be able to access the data residing at other sites.

- For instance, in a distributed banking system, where each branch stores data related to that branch, it is possible for a user in one branch to access data in another branch.

✔Autonomy.

- The primary advantage of sharing data by means of data distribution is that each site is able to retain a degree of control over data that are stored locally.

- In a centralized system, the database administrator of the central site controls the database.

- In a distributed system, there is a **global database administrator** responsible for the entire system.

- A part of these responsibilities is delegated to the local database administrator for each site.

- Depending on the design of the distributed database system, each administrator may have a different degree of **local autonomy.**

- **The possibility of local** autonomy is often a major advantage of distributed databases.

✔Availability.

- If one site fails in a distributed system, the remaining sites may be able to continue operating.

- In particular, if data items are **replicated in several** sites, a transaction needing a particular data item may find that item in any of several sites.

- Thus, the failure of a site does not necessarily imply the shutdown of the system.

- The failure of one site must be detected by the system, and appropriate action may be needed to recover from the failure.

- The system must no longer use the services of the failed site. Finally, when the failed site recovers or is repaired, mechanisms must be available to integrate it smoothly back into the system.

An Example of a Distributed Database

- Consider a banking system consisting of four branches in four different cities.

- Each branch has its own computer, with a database of all the accounts maintained at that branch.

- Each such installation is considered as a site.

- There also exists one single site that maintains information about all the branches of the bank.

- Each branch maintains (among others) a relation account(Account-schema), where

Account-schema = (account-number, branch-name, balance)

- The site containing information about all the branches of the bank maintains the relation ***branch(Branch-schema), where***

Branch-schema = (branch-name, branch-city, assets)

- There are other relations maintained at the various sites.

- To illustrate the difference between the two types of transactions—local and global—at the sites, consider a transaction to add $50 to account number A-150 located at the Valley view branch. If the transaction was initiated at the Valley view branch, then it is considered **local**; otherwise, it is considered global.

- A transaction to transfer $50 from account A-150 to account A-305, which is located at the Hillside branch, is **a global transaction**, since accounts in two different sites are accessed as a result of its execution.

Implementation Issues for Distributed Databases

- **Atomicity of transactions** is an important issue in building a distributed database system.

- If a transaction runs across two sites, unless the system designers are careful, it may commit at one site and abort at another, leading to an inconsistent state.

- The two-phase commit protocol (2PC) is most widely used.

- **Concurrency control** is another issue in a distributed database.

- Since a transaction may access data items at several sites, transaction managers at several sites may need to coordinate to implement concurrency control.

- If locking is used ,locking can be performed locally at the sites containing accessed data items, but there is also a possibility of deadlock involving transactions originating at multiple sites.

- Therefore deadlock detection needs to be carried out across multiple sites.

- The primary disadvantage of distributed database systems is the added complexity required to ensure **proper coordination among the sites.**

- **This increased complexity takes various forms:**

✔ **Software-development cost.** It is more difficult to implement a distributed database system; thus, it is more costly.

✔ **Greater potential for bugs.** Since the sites that constitute the distributed system operate in parallel, it is harder to ensure the correctness of algorithms, especially operation during failures of part of the system, and recovery from failures.

✔ **Increased processing overhead.** The exchange of messages and the additional computation required to achieve intersite coordination are a form of overhead that is not present in centralized systems.

❖ **Distributed databases are classified as,** i)Homogeneous and ii)Heterogeneous Databases.

i) **In a homogeneous distributed database,** all sites have identical database management system software, are aware of one another, and agree to cooperate in processing users' requests. (i.e. all sites share a common global schema and run the identical DBMS software).

ii) Heterogeneous distributed database:

- **In this different sites may use different** schemas, and different database management system software.
- The sites may not be aware of one another, and they may provide only limited facilities for cooperation in transaction processing.(i.e. system may be composed of a variety of data models for example relational, object oriented, hierarchical models etc.)

❑ Distributed Data Storage

- Consider a relation r that is to be stored in the database. There are two approaches to storing this relation in the distributed database:

 Replication. The system maintains several identical replicas (copies) of the relation, and stores each replica at a different site.

 Fragmentation. The system partitions the relation into several fragments, and stores each fragment at a different site.

- Fragmentation and replication can be combined: A relation can be partitioned into several fragments and there may be several replicas of each fragment.

Data Replication

- If relation r is replicated, a copy of relation r is stored in two or more sites. In the most extreme case, **a copy is stored in every site in the** system which is called as **full replication.**

- **Advantages**

- ***1)*Availability.**

- **If one of the sites containing relation r fails, then the relation r** can be found in another site. Thus, the system can continue to process queries involving r, despite the failure of one site.

- **2) Increased parallelism.** In the case where the majority of accesses to the relation r result in only the reading of the relation, then several sites can process queries involving r in parallel.

The more replicas of r there are, the greater the chance that the needed data will be found in the site where the transaction is executing. Hence, data replication minimizes movement of data between sites.

- **Disadvantages to replication:-**
- **Increased overhead on update.**

-The system must ensure that all replicas of a relation r are consistent; otherwise, erroneous computations may result.

-Thus, whenever r is updated, the update must be propagated to all sites containing replicas. The result is increased overhead.

- For example, in a banking system, where account information is replicated in various sites, it is necessary to ensure that the balance in a particular account should be same in all sites.

Data Fragmentation

- Division of relation r into fragments r_1, r_2, …, r_n which contain sufficient information to reconstruct relation r.
- **Horizontal fragmentation**: each tuple of *r* is assigned to one or more fragments.

- **Vertical fragmentation**: the schema for relation *r* is split into several smaller schemas
  - All schemas must contain a common candidate key (or superkey) to ensure lossless join property.
  - A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key.
- **Example :** relation account with following schema

  Account-schema = (*branch-name*, *account-number*, *balance*)

- In **horizontal fragmentation, a relation *r is partitioned into a number of subsets,*** *r1, r2, . . . , rn.*

- Each tuple of relation r must belong to at least one of the fragments, so that the original relation can be reconstructed, if needed.

- The account relation can be divided into several different fragments, each of which consists of tuples of accounts belonging to a particular branch.

- If the banking system has only two branches—Hillside and Valleyview—then there are two different fragments:

$$account1 = \sigma_{branch\text{-}name = \text{“Hillside”}}(account)$$
$$account2 = \sigma_{branch\text{-}name = \text{“Valleyview”}}(account)$$

Horizontal Fragmentation of *account* Relation

branch-name	*account-number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$$account_1 = \sigma_{branch\text{-}name = \text{“}Hillside\text{”}}(account)$$

branch-name	*account-number*	*balance*
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

$$\mathbf{account_2 = \sigma_{branch\_name = \text{“}Valleyview\text{”}}(account)}$$

- Horizontal fragmentation is usually used to keep tuples at the sites where they are used the most, to minimize data transfer.

- In general, a horizontal fragment can be defined as a **selection on the global relation r.**

- That is, we use a predicate P_i to construct fragment r_i:

$$r_i = \sigma_{P_i}(r)$$

- We reconstruct the relation r by taking the union of all fragments; that is,

$$r = r1 \cup r2 \cup \cdots \cup rn$$

- **Vertical fragmentation of *r(R) involves the definition of several subsets*** of attributes *R1, R2, . . .,Rn of the schema R so that,*

$$R = R1 \cup R2 \cup \cdots \cup Rn$$

- Each fragment ri of r is defined by $ri = \Pi_{Ri}(r)$.
- The fragmentation should be done in such a way that we can reconstruct relation *r* from the fragments by taking the natural join

$$r = r1 \bowtie r2 \bowtie r3 \quad \ldots\ldots\ldots rn$$

- One way of ensuring that the relation *r can be reconstructed is to include the* primary-key attributes of *R in each of the Ri.*

- More generally, any superkey can be used.

- It is often convenient to add a special attribute, called a *tuple-id, to the schema R.*

- *The tuple-id value of a tuple is a unique value that distinguishes the tuple from all* other tuples.

- The tuple-id attribute thus serves as a candidate key for the augmented(improved) schema, and is included in each of the *Ris*.

- *The physical or logical address for a tuple* can be used as a tuple-id, since each tuple has a unique address.

- To illustrate vertical fragmentation, consider a university database with a relation ***employee-info*** *that stores, for each employee, employee-id, name, designation, and salary.*

- For privacy reasons, this relation may be fragmented into a relation ***employee-privateinfo*** containing *employee-id and salary, and another relation* ***employee-public-info*** *containing* attributes *employee-id, name, and designation.*

- *These may be stored at different* sites, again for security reasons.
- The two types of fragmentation can be applied to a single schema; for instance, the fragments obtained by horizontally fragmenting a relation can be further partitioned vertically.

- Fragments can also be replicated. A fragment can be replicated, replicas of fragments can be fragmented further, and so on.

Vertical Fragmentation of Account-*info* Relation

branch-name	*customer-name*	*tuple-id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$deposit_1 = \Pi_{branch\text{-}name,\ customer\text{-}name,\ tuple\text{-}id}(\textbf{Account}\text{-}info)$

account number	*balance*	*tuple-id*
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

$deposit_2 = \Pi_{account\text{-}number,\ balance,\ tuple\text{-}id}(\textbf{Account}\text{-}info)$

Advantages of Fragmentation

- **Horizontal:**
  - allows parallel processing on fragments of a relation
  - allows a relation to be split so that tuples are located where they are most frequently accessed
- **Vertical:**
  - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
  - tuple-id attribute allows efficient joining of vertical fragments
  - allows parallel processing on a relation
- **Vertical and horizontal fragmentation can be mixed.**
  - Fragments may be successively fragmented to an arbitrary depth.

Data Transparency

- **Data transparency**: Degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system
- **Transparency issues :**
- **Fragmentation transparency :**Users are not required to know how a relation has been fragmented.
- **Replication transparency:** Users view each data object as logically unique. The distributed system may replicate an object to increase either system performance or data availability. Users do not have to be concerned with what data objects have been replicated, or where replicas have been placed.
- **Location transparency.** Users are not required to know the physical location of the data. The distributed database system should be able to find any data as long as the data identifier is supplied by the user transaction.

- Data items—such as relations, fragments, and replicas—must have unique names.

- In a distributed database, however, we must take care to ensure that two sites do not use the same name for distinct data items.

- One solution to this problem is to require all names to be registered in a central **name server.**

- The **name server** helps to ensure that the same name does not get used for different data items.

- We can also use the **name server** to locate a data item, given the name of the item.

Distributed Transactions

- Access to the various data items in a distributed system is usually accomplished through transactions, which must preserve the ACID properties.

- Two types of transactions are local and global transactions.

- The **local transactions are those** that access and update data in only one local database; the **global transactions are** those that access and update data in several local databases.

- However, for global transactions, this task is much more complicated, since several sites may be participating in execution.

- The failure of one of these sites, or the failure of a communication link connecting these sites, may result in erroneous computations.

Distributed Transactions

- **Transaction may access data at several sites.**
- Each site has a local **transaction manager** responsible for:
  - Maintaining a log for recovery purposes
  - Participating in coordinating the concurrent execution of the transactions executing at that site.
  - ensure the ACID properties of those transactions that execute at that site.

- Each site has a **transaction coordinator**, which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing sub transactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites.

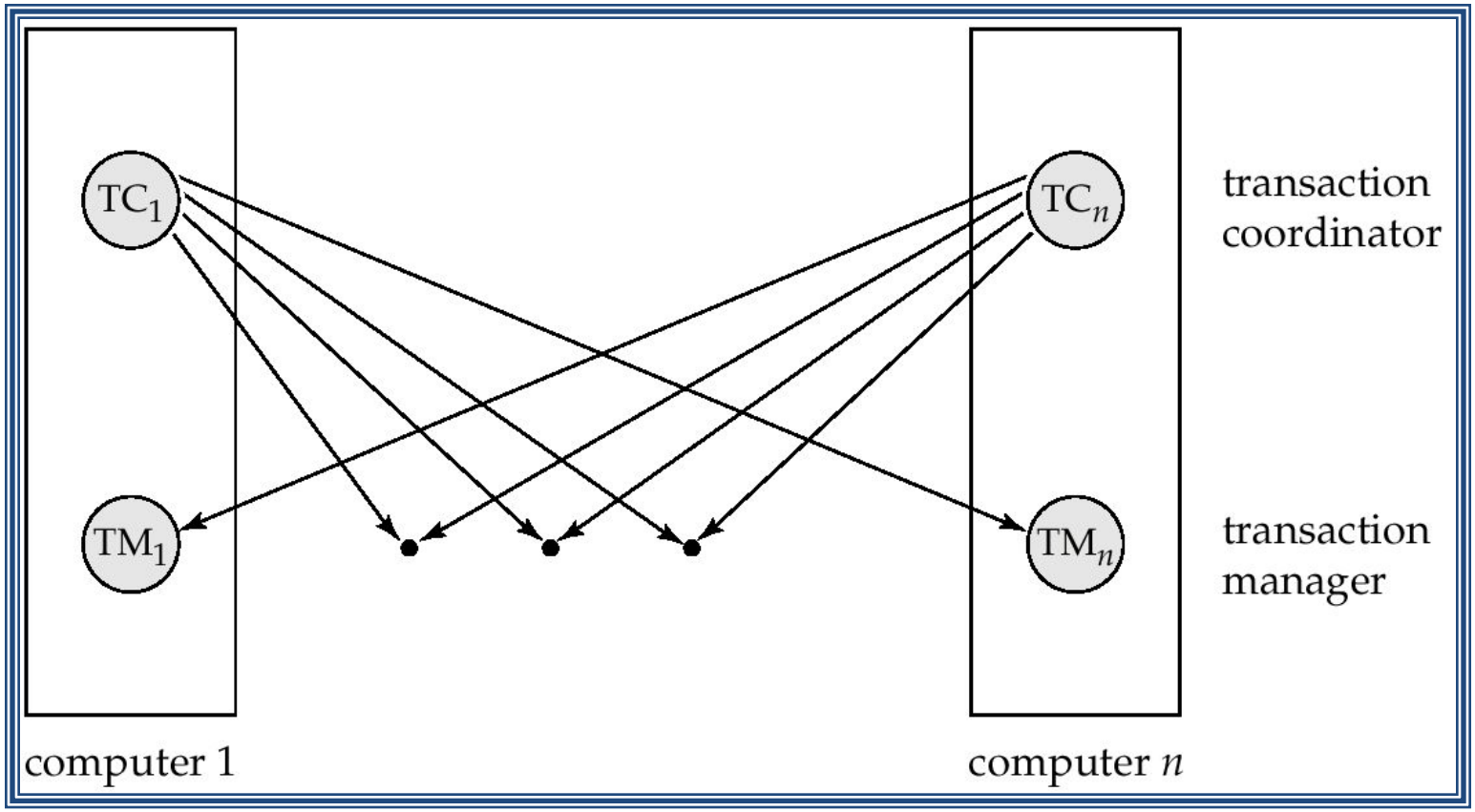
- The various transaction managers cooperate to execute global transactions.

- consider an abstract model of a transaction system, in which each site contains two subsystems:

  - *The* ***transaction manager*** ***manages the execution of those transactions (or subtransactions)*** that access data stored in a local site. Here that each such transaction may be either a local transaction or part of a global transaction.

  - *The* ***transaction coordinator*** ***coordinates the execution of the various transactions*** (both local and global) initiated at that site.

Transaction System Architecture



- The structure of a transaction manager is similar in many respects to the structure of a centralized system.

- Each **transaction manager** is responsible for

  --Maintaining a log for recovery purposes

  -- Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site.

- The transaction coordinator subsystem is not needed in the centralized environment, since a transaction accesses data at only a single site.

- For each such transaction, the **coordinator is responsible** for

  -- Starting the execution of the transaction

  -- Breaking the transaction into a number of subtransactions and distributing these subtransactions to the appropriate sites for execution.

  -- Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

System Failure Modes

- A distributed system may suffer from the same types of failure that a centralized system does (for example, software errors, hardware errors, or disk crashes).
- The basic failure types are,
  - **Failure of a site.**
  - **Loss of messages**
    - Handled by network transmission control protocols such as TCP-IP
  - **Failure of a communication link**
    - Handled by network protocols, by routing messages via alternative links
  - **Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that require any connection between them.

- The system uses transmission-control protocols, such as TCP/IP, to handle such errors.

- However, if two sites A and B are not directly connected, messages from one to the other must be routed through a sequence of communication links.

- If a communication link fails, messages that would have been transmitted across the link must be rerouted.

- In some cases, it is possible to find another route through the network, so that the messages are able to reach their destination.

- In other cases, a failure may result in there being no connection between some pairs of sites.

❖Commit Protocols

•Commit protocols are used to ensure atomicity across sites
-a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
-not acceptable to have a transaction committed at one site and aborted at another.

• T must either commit at all sites, or it must abort at all sites.

•To ensure this property, the transaction coordinator of T must execute a commit protocol.

•The two-phase commit (2 PC) protocol is widely used.

Two-Phase Commit

- Consider a transaction T initiated at site Si, where the transaction coordinator is Ci.
- When T completes its execution—that is, when all the sites at which T has executed inform Ci that T has completed—Ci starts the 2PC protocol.

• Phase 1. Ci adds the record<prepare T> to the log, and forces the log onto stable storage.

- It then sends a <prepare T> message to all sites at which T executed. On receiving such a message, the transaction manager at that site determines whether it is willing to commit its portion of T.

-**If the answer is no**, it adds a record <no T> to the log, and then responds by sending an **abort T** message to Ci.

-**If the answer is yes,** it adds a record <ready T> to the log, and forces the log (with all the log records corresponding to T) onto stable storage.

- The transaction manager then replies with a **ready T** message to Ci.

Phase 2. When Ci receives responses to the prepare T message from all the sites, or when a prespecified interval of time has elapsed since the prepare T message was sent out, Ci can determine whether the transaction T can be committed or aborted.

-Transaction T can be committed if Ci received a **ready T** message from all the participating sites.

-Otherwise, transaction T must be aborted. Depending on the decision, either a **record <commit T> or a record <abort T>** is added to the log and the log is forced onto stable storage.

-At this point, the outcome of the transaction has been sealed. Following this point, the coordinator sends either a commit T or an abort T message to all participating sites.

-When a site receives that message, it records the message in the log.

- A site at which T executed can unconditionally abort T at any time before it sends the message **ready T** to the coordinator.

- Once the message is sent, the transaction is said to be in the **ready state** at the site. The **ready T** message is, in effect, a promise by a site to follow the coordinator's order to commit T or to abort T.

- To give such assurance, the needed information must first be stored in stable storage.

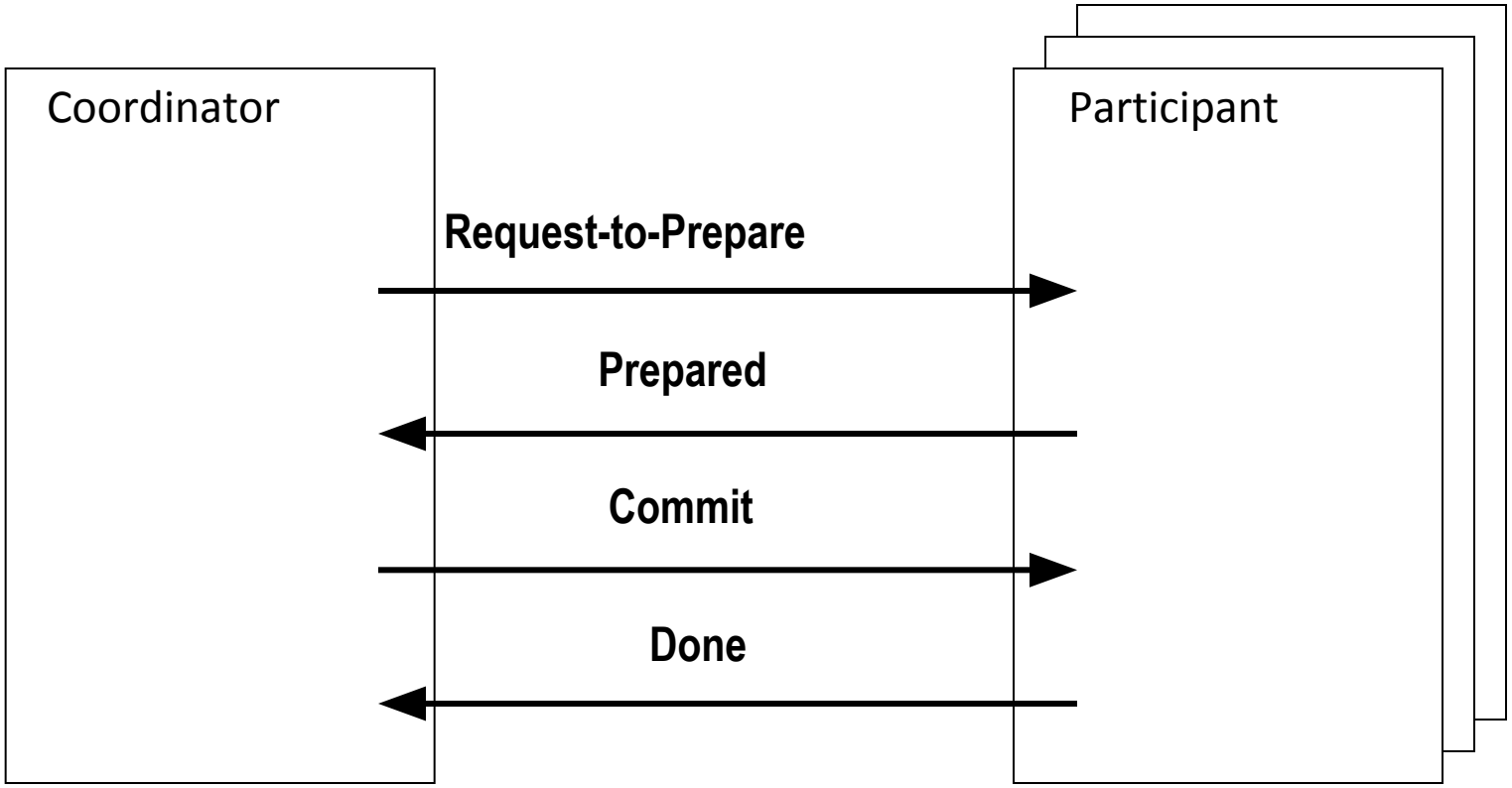
- Otherwise, if the site crashes after sending ready T, it may be unable to make good on its promise.

- Further, locks acquired by the transaction must continue to be held till the transaction completes.

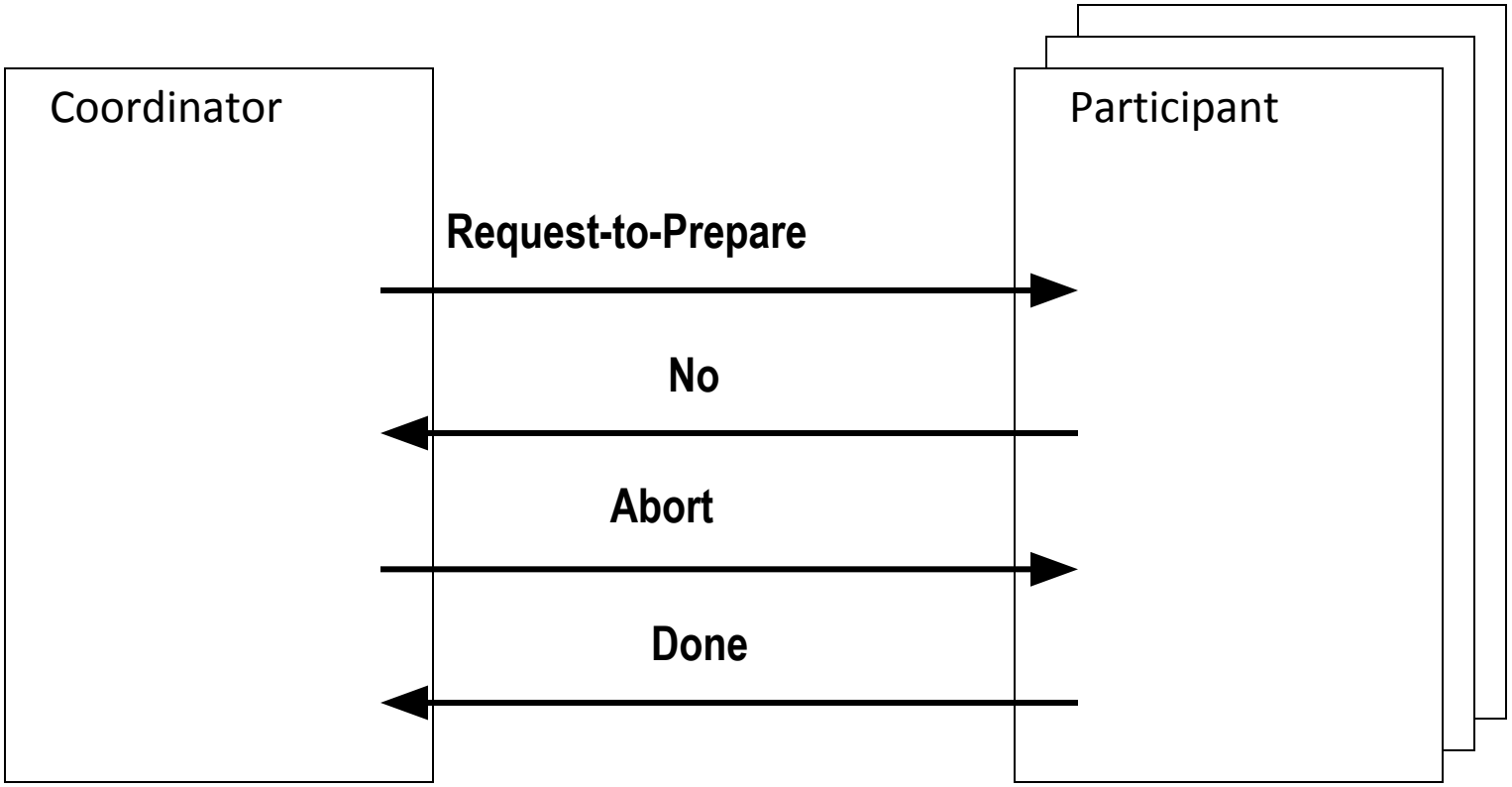
- In some implementations of the 2PC protocol, a site sends an acknowledge T message to the coordinator at the end of the second phase of the protocol.

- When the coordinator receives the acknowledge T message from all the sites, it adds the record **<complete T>** to the log.

Case 1: Commit



Case 2: Abort



Phase 1: Obtaining a Decision

- Coordinator asks all participants to *prepare* to commit transaction T_i.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **prepare** *T* messages to all sites at which *T* executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** *T*> to the log and send **abort** *T* message to C_i
  - if the transaction can be committed, then:
  - add the record <**ready** *T*> to the log
  - force *all records* for *T* to stable storage
  - send **ready** *T* message to C_i

Phase 2: Recording the Decision

- T can be committed of C_i received a **ready** T message from all the participating sites: otherwise T must be aborted.

- Coordinator adds a decision record, <**commit** T> or <a**bort** T>, to the log and forces record onto stable storage.

- Coordinator sends a message to each participant informing it of the decision (commit or abort).

- Participants take appropriate action locally.

Handling of Failures - Site Failure

The 2PC protocol responds in different ways to various types of failures:

- **Failure of a participating site.**

If the coordinator Ci detects that a site has failed, it takes following actions:

-If the site fails before responding with a **ready T** message to Ci, the coordinator assumes that it responded with an **abort T** message.

-If the site fails after the coordinator has received the **ready T** message from the site, the coordinator executes the rest of the commit protocol in the normal fashion, ignoring the failure of the site.

Handling of Failures - Site Failure

When site S_i recovers, it examines its log to determine the outcome of transactions active at the time of the failure.

- Log contain <**commit** T> record: site executes **redo** (T)
- Log contains <**abort** T> record: site executes **undo** (T)
- Log contains <**ready** T> record: site must consult C_i to determine the outcome of T.
  - If T committed, **redo** (T)
  - If T aborted, **undo** (T)

- The log contains no control records concerning T
  - implies that S_k failed before responding to the **prepare** T message from C_i
  - S_k must execute **undo** (T)

Handling of Failures- Coordinator Failure

- If the coordinator fails in the middle of the execution of the commit protocol for transaction T, then the participating sites must decide the outcome of T.

- In certain cases, the participating sites cannot decide whether to commit or abort T, and therefore these sites must wait for the recovery of the failed coordinator.

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for T is executing then participating sites must decide on T's outcome:
  1. If an active site contains a <**commit** T> record in its log, then T must be committed.
  2. If an active site contains an <**abort** T> record in its log, then T must be aborted.
  3. If some active participating site does not contain a <**ready** T> record in its log, then the failed coordinator C_i cannot have decided to commit T. Can therefore abort T.
  4. If none of the above cases holds, then all active sites must have a <**ready** T> record in their logs, but no additional control records (such as <**abort** T> of <**commit** T>). In this case active sites must wait for C_i to recover, to find decision.
- **Blocking problem** : active sites may have to wait for failed coordinator to recover.

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator.
- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
    - Again, no harm results

Three Phase Commit (3PC)

- Assumptions:
  - No network partitioning
  - At any point, at least one site must be up.
  - At most K sites (participants as well as coordinator) can fail
- **Phase 1: Obtaining Preliminary Decision: Identical to 2PC Phase 1.**
  - Every site is ready to commit if instructed to do so
  - Under 2 PC each site is obligated to wait for decision from coordinator
  - Under 3PC, knowledge of pre-commit decision can be used to commit despite coordinator failure.
  - The protocol avoids blocking by introducing an extra third phase where multiple sites are involved in the decision to commit.

Phase 2. Recording the Preliminary Decision

- Coordinator adds a decision record (<**abort** *T*> or < **precommit** *T*>) in its log and forces record to stable storage.
- Coordinator sends a message to each participant informing it of the decision
- Participant records decision in its log
- If abort decision reached then participant aborts locally
- If pre-commit decision reached then participant replies with <**acknowledge** *T*>

Phase 3. Recording Decision in the Database

Executed only if decision in phase 2 was to precommit

- Coordinator collects acknowledgements. It sends <**commit** *T*> message to the participants as soon as it receives K acknowledgements.
- Coordinator adds the record <**commit** *T*> in its log and forces record to stable storage.
- Coordinator sends a message to each participant to <**commit** *T*>
- Participants take appropriate action locally.

Handling Site Failure

- **Site Failure.** Upon recovery, a participating site examines its log and does the following:
  - Log contains <**commit** *T*> record: site executes **redo** (*T*)
  - Log contains <**abort** *T*> record: site executes **undo** (*T*)
  - Log contains <**ready** *T*> record, but no <**abort** *T*> or <**precommit** *T*> record: site consults C_i to determine the outcome of *T*.

    -if C_i says *T* aborted, site executes **undo** (*T*) (and writes <**abort** *T*> record)

    -if C_i says T committed, site executes **redo** (*T*) (and writes < **commit** *T*> record)

Handling Site Failure (Cont.)

  - if *C* says *T* committed, site resumes the protocol from receipt of **precommit** *T* message (thus recording <**precommit** *T*> in the log, and sending **acknowledge** *T* message sent to coordinator).

- Log contains no <**ready** *T*> record for a transaction *T*: site executes **undo** (*T*) writes <**abort** *T*> record.

Coordinator – Failure Protocol

1. The active participating sites select a new coordinator, C_{new}
2. C_{new} requests local status of T from each participating site
3. Each participating site including C_{new} determines the local status of T:
   - **Committed**. The log contains a < **commit** T> record
   - **Aborted**. The log contains an <**abort** T> record.
   - **Ready**. The log contains a <**ready** T> record but no <**abort** T> or <**precommit** T> record
   - **Precommitted**. The log contains a <**precommit** T> record but no <**abort** T> or <**commit** T> record.
   - **Not ready**. The log contains neither a <**ready** T> nor an <**abort** T> record.

A site that failed and recovered must ignore any **precommit** record in its log when determining its status.

4. Each participating site records sends its local status to C_{new}

Coordinator Failure Protocol (Cont.)

5. C_{new} decides either to commit or abort T, or to restart the three-phase commit protocol:
   - Commit state for any one participant ⇒ commit
   - Abort state for any one participant ⇒ abort.
   - Precommit state for any one participant and above 2 cases do not hold ⇒

     A precommit message is sent to those participants in the uncertain state. Protocol is resumed from that point.

END

References

- https://www.db-book.com › slide-dir
- Book-Database **System** Concepts.

Distinct characteristics of C/S

- Client-server is a computing architecture which separates a client from a server
- It is almost always implemented over a computer network
- The most basic type of client-server architecture employs only two types of nodes: clients and servers.
  - This type of architecture is sometimes referred to as *two-tier*.
  - It allows devices to share files and resources.
- Server provides the service
- Client is considered as the customer requesting the service

- The server service can be shared among a number of clients
- Clients must request or initiate the service
- The location of the server in the network is transparent to clients
- Transaction between C/S is message-passing based
- C/S architecture is scalable
  - horizontally (more clients can added)
  - Vertically (more servers can be added)
- The server is centrally maintained whereas clients are independent of each other

Types of Servers

- Application Servers
- Audio/Video Servers
- Chat Servers
- Fax Servers
- FTP Servers
- Groupware Servers
- List Servers
- Mail Servers
- News Servers
- Proxy Servers
- Telnet Servers
- Web Servers
- Z39.50 Servers

N-tier architecture

- N-Tier architecture
  - is an industry-proved software architecture model,
  - suitable to support enterprise-level client/server applications by resolving issues like scalability, security, fault tolerance and etc.

Three layers

- **Presentation layer**
  - a layer that users can access directly, such as desktop UI, web page etc. Also called client.
- **Application layer**
  - this layer encapsulates the business logic (such as business rules and data validation), domain concept, data access logic etc. Also called middle layer.
- **Data layer**
  - the external data source to store the application data,
    - such as database server, ERP system, mainframe or other legacy systems etc.

1, 2, 3 or More Tier Architecture

- **1-Tier**:
  - all above layers can only run in one computer.
  - In order to achieve 1-Tier, we need to use the embedded database system, which cannot run in an individual process.
    - Otherwise, there will be at least 2-Tier because non-embedded databases usually can run in an individual computer (tier).
- **2-Tier**:
  - either presentation layer and application layer can only run in one computer,
  - or application layer and data layer can only run in one computer.
  - The whole application cannot run in more than 2 computers.

- **3-Tier**:
  - the simplest case of N-Tier architecture;
  - all above three layers are able to run in three separate computers.
  - Practically, these three layers can also be deployed in one computer (3-Tier architecture, but deployed as 1-Tier).
- **N-Tier**:
  - 3 or more tiers architecture.
  - Some layers in 3-Tier can be broken further into more layers.
    - These broken layers may be able to run in more tiers.
    - For example,
      - application layer can be broken into business layer, persistence layer or more.
      - Presentation layer can be broken into client layer and client presenter layer.
  - In diagram, in order to claim a complete N-Tier architecture,
    - client presenter layer, business layer and data layer should be able to run in three separate computers (tiers).
    - Practically, all these layers can also be deployed in one compute (tier).

- **Client layer**: this layer is involved with users directly. There may be several different types of clients coexisting, such as Window form, HTML web page etc.

- **Client presenter layer**: contains the presentation logic needed by clients, such as ASP .NET MVC in IIS web server. Also it adapts different clients to the business layer.

- **Business layer**: handles and encapsulates all of business domains and logics; also called domain layer.

- **Persistence layer**: handles the read/write of the business data to the data layer, also called data access layer (DAL).

- **Data layer**: the external data source, such as a database.

Advantages and Disadvantages of 1 or 2-Tier Architecture

- **Advantages**:
  - simple and fast for a lower number of users
    - due to fewer processes and fewer tiers;
  - low cost for hardware, network, maintenance and deployment
    - due to less hardware and network bandwidth needed.

- **Disadvantages**:
  - will have issues when the number of users gets big;
  - has limitation to solve issues like security, scalability, fault tolerance and etc
    - because it can be deployed in only 1 or 2 computers.

N-Tier Architecture - Advantages

- Scalable:
  - this is due to its capability of multiple tier deployment .
  - For example,
    - the data tier can be scaled up by database clustering without other tiers involving.
    - The web client side can be scaled up by load-balancer easily without affecting other tiers.
    - Windows server can be clustered easily for load balancing
    - In addition, business tier server can also be clustered to scale up the application, such as Weblogic cluster in J2EE.

- Better and finer security control to the whole system:
  - we can enforce the security differently for each tier if the security requirement is different for each tier.
  - For example,
    - business tier and data tier usually need higher security level than presentation tier does, then we can put these two high security tiers behind firewall for protection.
    - 1 or 2 tiers architecture cannot fully achieve this purpose because of a limited number of tiers.
    - Also, for N-Tier architecture, users cannot access business layer and data layer directly, all requests from users are routed by client presenter layer to business layer, then to data layer. Therefore, client presenter layer also serves as a proxy-like layer for business layer, and business layer serves as a proxy-like layer for data layer. These proxy-like layers provides further protection for their layers below.

- Better fault tolerance ability:
  - for example,
    - the databases in data layer can be clustered for failover or load balance purpose without affecting other layers.

- Independent tier upgrading and changing without affecting other tiers:
  - in object-oriented world, Interface-dependency implementation can decouples all layers very well so that each layer can change individually without affecting other layers too much.
  - Interface-dependency means a layer depends on another layer by interfaces only.

- Friendly and efficient for development:
  - the decoupled layers
    - are logic software component groups mainly by functionality,
    - are very software development friendly and efficient.
  - Each layer
    - can be assigned individually to a team who specializes in the specific functional area; a specialized team can handle the relevant task better and more efficiently.

- Friendly for maintenance:
  - N-Tier architecture groups different things together mainly by functionality and then makes things clear, easily understandable and manageable.

- Friendly for new feature addition:
  - due to the logical grouped components and the decoupling brought by N-Tier architecture, new features can be added easily without affecting too much on the whole system.

- Better reusability:
  - due to the logically grouped components and the loose couplings among layers.
    - Loosely-coupled component groups are usually implemented in more general ways, so they can be reused by more other applications.

The Disadvantages of the N-Tier Deployment

- The performance of the whole application
  - may be slow if the hardware and network bandwidth aren't good enough because more networks, computers and processes are involved.

- More cost for hardware, network, maintenance and deployment
  - because more hardware and better network bandwidth are needed.