DISTRIBUTED DATABASES

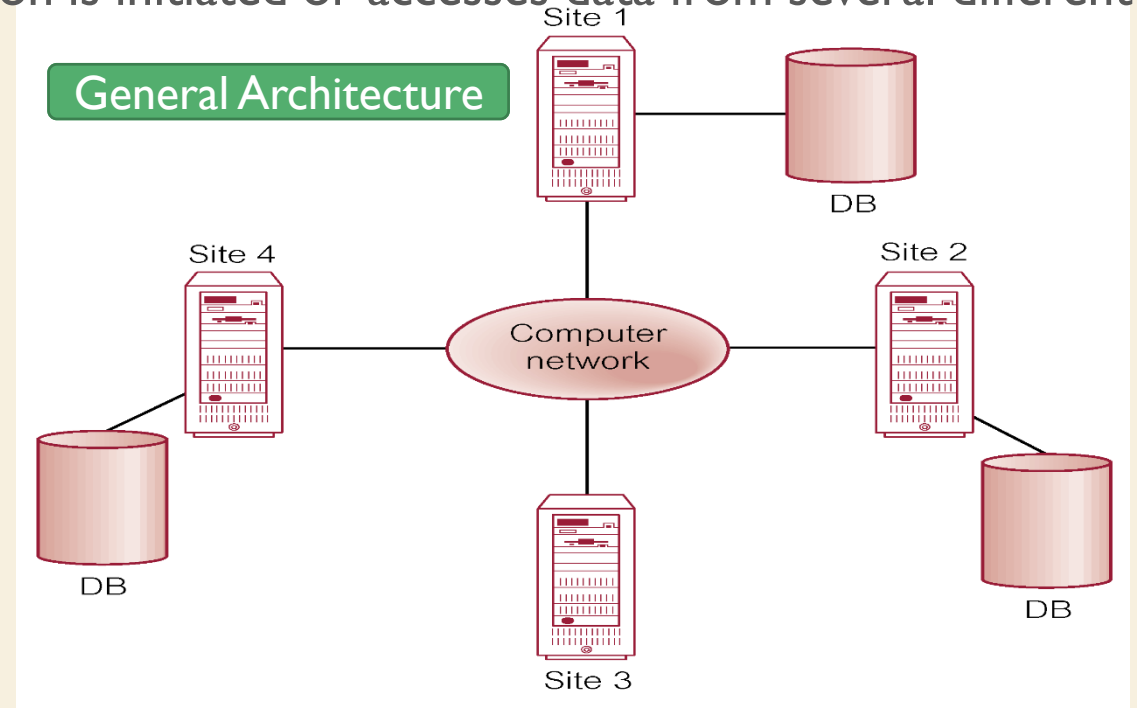
COMPILED BY
M ABDUL JAWAD

INTRODUCTION TO DISTRIBUTED DATABASES

- A distributed database system is a one where the database is stored on several computers. The computers in a distributed system communicate with one another through various communication media – High Speed Private Networks or the Internet.
- Distributed Databases don't share main memory or disks.
- The computers in a distributed system are referred by a number of different names such as Sites or Nodes.
- The main difference of distributed databases with conventional ones is, distributed databases are typically geographically separated & separately administered having slower interconnection.

DISTRIBUTED DATABASES

- We also differentiate in distributed databases the – local and global transactions.
- **Local Transaction**: A local transaction is one that accesses data only from sites where the transaction was initiated.
- **Global Transaction**: In the case of Global Transactions, the data is either accessed from the site where transaction is initiated or accesses data from several different sites.



NEED FOR DISTRIBUTED DATABASES

- Several reasons justify the need of implementing distributed databases:
  - **Sharing Data:** The major advantage being having a provision of an environment where users at one site may be able to access data residing at other sites.
    - Example Distributed University Campuses.
  - **Autonomy:** Unlike a centralized approach where everything is controlled from a central location, data distribution is such that each site is able to retain a degree of control over data that are stored locally. In a distributed database we do have a database administrator responsible for the entire system, but part of these responsibilities is delegated to local database administrator for each site.

  Each database administrator has a different degree of local autonomy – depending on the design of distributed databases.

NEED FOR DISTRIBUTED DATABASES

- **Availability:** If one site fails in a distributed system, the remaining sites may be to continue operating. In particular, if data items are replicated in several sites, the failure of a single site doesn't necessarily mean the shutdown of a system.
- Some of **disadvantages** include:
  - **Complexity**
  - **Cost**
  - **Security**
  - **Lack of standards**
  - **Lack of experience**
  - **Database design more complex**

TYPES OF DISTRIBUTED DATABASES

- Distributed databases can be either ***Homogeneous Distributed Databases*** or ***Heterogeneous Distributed Databases.***
- **Homogenous Distributed Databases**: In homogenous distributed database systems, all sites have identical database management software, sites are aware of one another and agree to cooperate in processing users requests.
  - In such systems local sites surrender a portion of their autonomy in terms of their right to change schemas or database management system software. But the software must be such it must itself cooperate with other sites in exchanging information about transactions.

TYPES OF DISTRIBUTED DATABASES

- **Heterogeneous Distributed Databases**: In Heterogeneous distributed databases different sites may use different schemas and different database management software. The sites may not be aware of one other and they may provide only limited facilities for cooperation in transaction processing.
- This difference in schemas are often a major problem for query processing while divergence in software becomes a hindrance for processing transactions that access multiple sites.

DISTRIBUTED DATA STORAGE

- We have 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 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

- Taking the same relation ***'r' ,*** a copy of relation ***'r'*** is stored in two or more sites. In most extreme case, we have ***full replication*** in which every copy is stored in every site in the system.
  - ***Advantages and Disadvantages of Replication:***
  - ***Availability:- If one of the sites containing relation 'r' fails, then the relation 'r' can be found in another site. This way system can continue to process queries involving 'r', despite the single site failure.***
  - ***Increased Parallelism:- In the case where majority of accesses to the relation 'r' result in only reading the relation then several sites can process queries involving 'r' in parallel. This way data replication minimizes movement of data between sites.***

DATA REPLICATION

- Increased overhead on update: The system must ensure that all replicas of a relation ‘r’ are consistent; otherwise, erroneous computations may result. So whenever, ‘r’ is updated, the update must be propagated to all sites containing replicas. This is an increased overhead.
  - In general, replication enhances performance of read operations and increases the availability of data to read-only transactions. However update transactions incur greater overhead.
  - We have a concept of **Primary Copy.**
  - A primary copy of a relation ***‘r’*** is one which is associated with a site where it was first created.

DATA FRAGMENTATION

- Data fragmentation involves dividing an original relation 'r' into a number of fragments $r_1, r_2, \ldots, r_n$ which contain sufficient information to reconstruct relation r.
- We have two schemes of fragmenting a relation:
  - Horizontal Fragmentation.
  - Vertical Fragmentation.

HORIZONTAL FRAGMENTATION

- **Horizontal fragmentation** splits the relation by assigning each tuple of ***'r'*** to one or more fragments.
- We have a relation r which is partitioned into a number of subsets – n subsets.
- Each tuple of relation must belong to at least one or more fragments so that original relation can be reconstructed, if needed.
- Example : relation account with following schema
- *Account = (account_number, branch_name , balance)*
- Horizontal fragmentation is usually used to keep tuples at the sites where they are used the most, to minimize data transfer.

HORIZONTAL FRAGMENTATION

account_number	branch_name	balance
A-305	Hillside	500
A-226	Hillside	336
A-155	Hillside	62

$$account_1 = \sigma_{branch\_name="Hillside"}(account)$$

account_number	branch_name	balance
A-177	Valleyview	205
A-402	Valleyview	10000
A-408	Valleyview	1123
A-639	Valleyview	750

$$account_2 = \sigma_{branch\_name="Valleyview"}(account)$$

In general, horizontal fragment is a selection on global relation r.

VERTICAL FRAGMENTATION

- Vertical Fragmentation is same as decomposition.Vertical Fragmentation of r ® involves definition of several subsets of attributes R1, R2, ….., Rn.
-

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

$deposit_1 = \Pi_{branch\_name,\ customer\_name,\ tuple\_id}\ (employee\_info\ )$

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

$deposit_2 = \Pi_{account\_number,\ balance,\ tuple\_id}\ (employee\_info\ )$

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
- **Vertical and horizontal fragmentation can be mixed.**
  - Fragments may be successively fragmented to an arbitrary depth.
- **Replication and fragmentation can be combined**
  - Relation is partitioned into several fragments: system maintains several identical replicas of each such fragment.

TRANSPARENCY

- ***Transparency*** is user of the distributed database should not be required to know where the data are physically located nor how the data can be accessed at the specific local site.
- Data Transparency can take several forms:
  - **Fragmentation Transparency**:
    - Users are not required to know how a relation has been fragmented.
  - **Replication Transparency:**
    - Users view to data is always unique, but for various constraints same data may be replicated at different sites. Users don't need to be concerned of where data objects have been replicated and placed.
  - **Location Transparency:**
    - Users aren't required to know the physical location of data. The distributed database should be able to find any data as long as data identifier is supplied by user transactions.

DATA ITEMS AND NAMING

- Data items in databases are Relations, Fragments and Replicas. These Data items must have unique names. That is – In distributed database environment we must take care to ensure that two sites don't use same name for distinct data items.
- Solution to this problem is Use of a ***registered central name server.***
  - ***Name Server helps to ensure that same name doesn't get used for different data items.***
  - ***This approach however has several drawbacks:***
    - ***First the name server may become a performance bottleneck when data items are located by their names, resulting in poor performance.***
    - ***Second, if the name server crashes, it may not be possible for any site in the distributed system to continue to run.***

DATA ITEMS AND NAMING

- The second approach uses a mechanism – ***Each site prefixes its own site identifier to any name that it generates.*** Although the approach ensures no two sites generate the same name.
  - This solution, however, fails to achieve location transparency – Given site identifiers are attached to names.
    - Examples: ***site17.account or account@site17.***
  - ***To address this problem, the database system can create a set of alternative names or aliases, for data items.*** A user may hence refer to data items by simple names that are translated by the system to complete names.
  - ***Plus users will be unaffected if the database administrator decides to move a data item form one site to another.***

DISTRIBUTED TRANSACTIONS

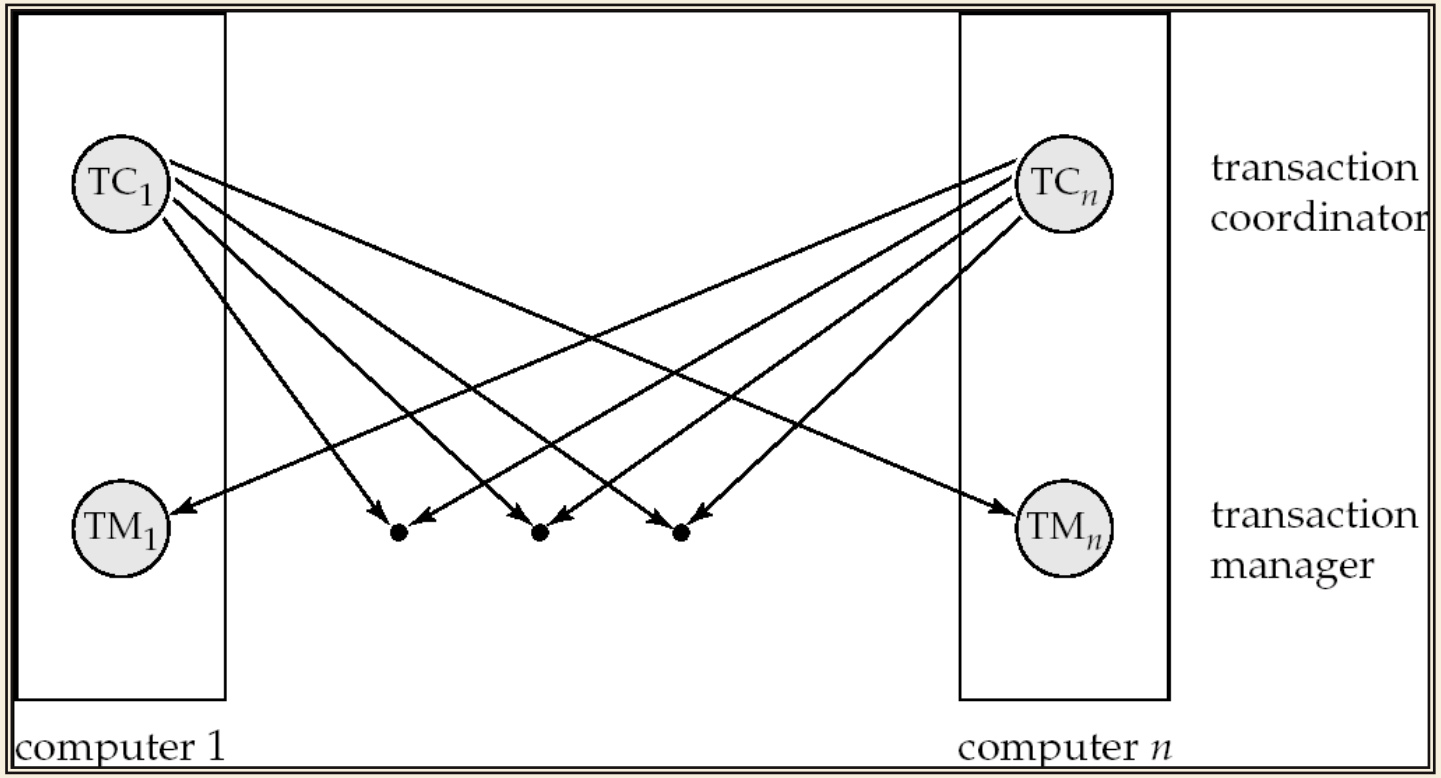
- Access to the various data in a distributed system is usually accomplished through transactions, which must preserve the ACID properties.
- Given the distributed database environment we have two types of transactions:
  - Local Transaction.
  - Global Transaction.
  - **Ensuring ACID properties of the local transactions isn't any issue however achieving ACID properties for Global Transactions is a tedious and complicated process as failure of communication link is obvious in distributed environment.**

SYSTEM STRUCTURE

Transaction may access data at several sites and each site contains two sub-systems:

- **Transaction Manager:**
  - Each site has its own ***local transaction manager***, whose function is to ensure ACID properties of those transactions that execute at that site and various transaction managers cooperate with each other to manage ***Global Transactions***.
  - Maintaining a log for recovery purposes
  - Participating in coordinating the concurrent execution of the transactions executing at that site.
- **Transaction Coordinator:**
  - Starting the execution of the transaction.
  - Breaking the transaction into a number of sub-transactions and distributing these sub-transactions to 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.

SYSTEM STRUCTURE



SYSTEM FAILURE MODES

- A distributed system may suffer from the basic types of failures like ***Software Errors, Hardware Errors or Disk Crashes.***
- However, the one that are more alarming are:
  - **Failure of Site**.
  - **Loss of Messages**.
    - Loss of messages is always a possibility in a distributed system. The system uses Transmission Control Protocols such as TCP/IP to handle such errors
  - **Failure of 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 lack any connection between them

COMMIT PROTOCOLS

- Commit protocols are used to ensure **atomicity** across sites.
- In a local database system, for committing a transaction, the transaction manager has to only convey the decision to commit to the recovery manager.
- However, in a distributed system, the transaction manager should convey the decision to commit to all the sites taking part in the transaction.
  - When processing is complete at each site, it reaches the ***partially committed transaction state*** and waits for all other transactions to reach their partially committed states. When it receives the message that all the sites are ready to commit, it starts to commit – ***The Global Commit.***
- ***One Phase Commit.***
- ***Two Phase Commit.***
- ***Three Phase Commit.***

TWO PHASE COMMIT

- Two Phase Commit takes into assumption – ***the Fail-Stop Model.***
  - ***Failed States simply stop working and don't send an incorrect set of messages.***
  - **Let *T* be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i**

- Two Phase Commit operates in two phases:
  - Phase 1 or Prepare Phase.
  - Phase 2 or Commit/Abort Phase or Decision Phase.
  - Transaction T completes its execution – all sites at which T has executed inform C_i *that has completed* – C_i *starts the 2PC protocol.*

TWO PHASE COMMIT – PHASE 1

- **Phase 1 : Prepare Phase**
- 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 where *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.

TWO PHASE COMMIT – PHASE 2

- **Phase 2 / Decision Phase / Commit/Abort Phase:**
  - *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. Once the record is recorded on stable storage it is irrevocable (even if failures occur)
  - Coordinator sends a message to each participant informing it of the decision (commit or abort)
  - Participants take appropriate action locally.
- 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 protocol.
- When the coordinator receives the acknowledge T message from all the sites, it adds the record ***<complete T>*** to the log.

HANDLING OF FAILURES

- 2PC responds in different ways to various types of failures:
- ***Failure of a Participating Site:-*** If the coordinator detects that the 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 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 commit protocol in the normal fashion, ignoring the failure of the site.
- When site S_i recovers, it examines its log to determine the fate 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*)

HANDLING OF FAILURES

- Log contains <**ready** *T*> record: site must consult C_i to determine the fate of *T*.
  - If C_i *is up, it notifies* S_k regarding whether T committed or aborted.
    - If *T* committed, **redo** (*T*)
    - If *T* aborted, **undo** (*T*)
  - If C_i *is down,* S_k must try to find the fate of T from other sites. It does so by sending querystatus T message to all sites in the system. On receiving such message a site must consult its log whether T has executed there, if Yes ,it must notify S_k about the outcome.
  - If no site has information regarding T, S_k must wait until any site recovers and coveys the outcome.
- 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

- **Failure of Coordinator:**
- If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*'s fate:
  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.

NETWORK PARTITION

- Given we are in a distributed environment a network failure is quite obvious leading to a situation known as Network Partition.
- When a Network Partitions, two possibilities exist;
  - **The coordinator and all its participants remain in one partition. So failure has no effect on the commit protocol.**
  - **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

RECOVERY AND CONCURRENCY CONTROL

- We need to deal with **In-doubt transactions** – Transactions in which have a <**ready** *T*>, but neither a <**commit** *T*>, nor an <**abort** *T*> log record is found.
  - The recovering site must determine the commit-abort status of such transactions by contacting other sites using various alternatives that handle failures – but this a **slow process and can block recovery process.**
- To circumvent this problem, the recovery algorithms typically provide support for noting **lock** information in the log.
  - Instead of <**ready** *T*>, write out <**ready** *T*, *L*> *L* = list of locks held by *T* when the log is written.
  - For every in-doubt transaction *T*, all the locks noted in the <**ready** *T*, *L*> log record are reacquired.
- After lock reacquisition, transaction processing can resume; the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions.

ALTERNATIVE MODELS OF TRANSACTION PROCESSING

- **Persistent Messaging**
  - Starting with the **funds transfer** by a bank check..
  - Persistent messages are the messages that are guaranteed to be delivered to the receipt exactly once (neither less nor more) regardless of the failures, if the transaction sending the message commits and are guaranteed not to be delivered if the transaction aborts.
  - Error handling is more complicated with persistent messaging than with 2 Phase Commit. - If the account where the check is to be deposited has been closed, the check must be sent back to the originating account and credited back.
  - Both sites must be provided with error-handling code, along with code to handle persistent messages.
- So we left it to requirement of an organization whether to implement a 2 Phase Commit or eliminate blocking by putting an extra effort in implementing Persistent Messaging.

IMPLEMENTATION OF PERSISTENT MESSAGING

- Persistent messaging can be implemented on top of an unreliable messaging infrastructure, which may lose messages or deliver them multiple times.
- **Sending Site Protocol**:
  - When a transaction wishes to send a persistent message, it writes a record containing the message in a special relation ***messages_to_send.*** Instead of directly sending out the message. This message is given a unique message identifier.
  - *The message delivery process monitors the relation and when a new message is found, it sends the message to destination. The concurrency control mechanism ensures that system process reads the message only after the transaction that wrote the message commits.*
  - *The message delivery process deletes a message from the relation only after it receives an acknowledgment from the destination site. If it receives no acknowledgement from the destination site, after some time it sends the message again. It repeats this until an acknowledgment is received. In case of permanent failures, the system will decide, after some period of time,that the message is undeliverable.*

IMPLEMENTATION OF PERSISTENT MESSAGING

- **Receiver Site Protocol:**
  - When a site receives a persistent message, it runs a transaction that adds the message to a special *received messages* relation, provided it is not already present in the relation (the unique message identifier allows duplicates to be detected).
  - After the transaction commits, or if the message was already present in the relation, the receiving site sends an acknowledgment back to the sending site.
  - Multiple deliveries of the message. In many messaging systems, it is possible for messages to get delayed arbitrarily, although such delays are very unlikely. Therefore, to be safe, the message must never be deleted from the *received messages* relation. Deleting it could result in a duplicate delivery not being detected

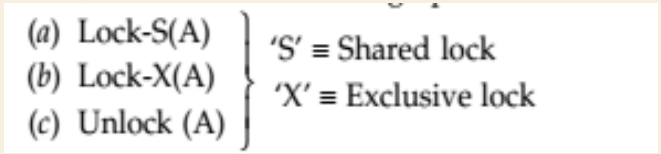
CONCURRENCY CONTROL IN DISTRIBUTED DATABASES

- Concurrency control is the process of managing simultaneous execution of transactions (like queries, updates, inserts, deletes etc.) in a multiprocessing database system without having them interfere with one another. ***This property of DBMS allows many transactions to access the same database at the same time without interfering with each other.***

- Concurrency control is important because simultaneous execution of transactions over shared database can create several data integrity and consistency problems.

LOCKING PROTOCOLS

- A **lock** is defined as a variable associated with a data item that describes the status of the item with respect to the possible operations that can be applied to it. Generally there is one lock for each data item in the database.
- Locks can be either:
  - Binary Locks.
    - A binary lock has two states—locked or unlocked. So, if 'A' is a data-item, we refer to the current value of the lock associated with item A as lock (A).
    - If Lock (A) = 1 then item-A cannot be accessed.
    - Else if Lock (A) = 0 then item-A can be accessed.
  - Share/Exclusive R/W locks.



SINGLE LOCK MANAGER APPROACH

- System maintains a *single* lock manager that resides in a *single* chosen site, say S_i
- All lock and unlock requests are made at site S_i.
- When a transaction needs to lock a data item, it sends a lock request to S_i and lock manager determines whether the lock can be granted immediately
  - If yes, lock manager sends a message to the site which initiated the request
  - If no, request is delayed until it can be granted, at which time a message is sent to the initiating site
- The transaction can read the data item from any one of the sites at which a replica of the data item resides. In case of a write, all the sites where a replica of the data item resides must be involved in the writing.

SINGLE LOCK MANAGER APPROACH

- **Advantages**
  - **Simple Implementation**: System requires two messages for handling lock requests and unlock requests.
  - **Simple Deadlock handling:** Since all lock and unlock requests are made at one site, the deadlock-handling algorithms can be applied directly.
- **Disadvantages**
  - **Bottleneck:** lock manager site becomes a bottleneck
  - **Vulnerability:** *If the site* S_i *Fails, the concurrency controller is lost. Either processing must stop, or a recovery scheme must be used so that a backup site can take over lock management from* S_i

DISTRIBUTED LOCK MANAGER

- **Distributed lock manager** is a compromise approach between the advantages and disadvantages of single lock manager approach.
- In distributed-lock-manager approach the lock manager approach function is distributed over several sites.
- Each site maintains a local lock manager whose function is to administer lock and unlock requests for those data items that are stored in that site.
- When a transaction wishes to lock a data item Q that is not replicated and resides at site S_i *a message is sent to the lock manager at site* S_i *requesting a lock.*
- **Advantage:** ***work is distributed and can be made robust to failures***
- **Disadvantage:** ***deadlock detection is more complicated***
- When the data is replicated alternate approaches need to be implemented ….

PRIMARY COPY

- Choose one replica of data item to be the **primary copy**.
  - Site containing the replica is called the **primary site** for that data item
  - Different data items can have different primary sites
- When a transaction needs to lock a data item *Q*, it requests a lock at the primary site of *Q*.
  - Implicitly gets lock on all replicas of the data item
- **Benefit**
  - *Concurrency control for replicated data handled similarly to un-replicated data - simple implementation.*
- **Drawback**
  - *If the primary site of Q fails, Q is inaccessible even though other sites containing a replica may be accessible*.

MAJORITY PROTOCOL

- Local lock manager at each site administers lock and unlock requests for data items stored at that site.
- When a transaction wishes to lock an un-replicated data item Q residing at site S_i, a message is sent to S_i 's lock manager.
  - If Q is locked in an incompatible mode, then the request is delayed until it can be granted.
  - When the lock request can be granted, the lock manager sends a message back to the initiator indicating that the lock request has been granted.

MAJORITY PROTOCOL

- In case of replicated data
  - If *Q* is replicated at n sites, then a lock request message must be sent to more than half of the n sites in which *Q* is stored.
  - The transaction does not operate on *Q* until it has obtained a lock on a majority of the replicas of *Q*.
  - When writing the data item, transaction performs writes on *all* replicas.
- Benefits
  - Can be used even when some sites are unavailable i.e. a site failure.
- However the approach suffers from major drawbacks:
  - Requires 2(*n*/2 + 1) messages for handling lock requests, and (*n*/2 + 1) messages for handling unlock requests.
  - Prone to deadlocks.

BIASED PROTOCOL

- The biased protocol is another approach to handle replication. The difference from the majority protocol is that requests for shared locks are given more favourable treatment than requests for exclusive locks.
- **Shared locks**. When a transaction needs to lock data item *Q*, it simply requests a lock on *Q* from the lock manager at one site containing a replica of *Q*.
- **Exclusive locks**. When transaction needs to lock data item *Q*, it requests a lock on *Q* from the lock manager at all sites containing a replica of *Q*.
- Advantage - imposes less overhead on **read** operations.
- Disadvantage - additional overhead on writes

TIMESTAMP BASED PROTOCOLS

- This protocol works on a principle that—"we must have a prior knowledge about the order in which the transactions will be accessed".

If a transaction (T_i) has been assigned timestamp, TS(T_i), and a new transaction, T_j, enters the system, then

$$TS(T_i) < TS(T_j)$$

Scheme Implementation Methods : There are two common methods for its implementation :

1. Using the **value of the system clock as the timestamp**. That is,

   Transaction's timestamp value = Value of the system clock.

   When the transaction enters the system.
2. Using a **logical counter** that is incremented after a new timestamp has been assigned. That is,

   Transaction's timestamp value = Value of the counter

   when the transaction enters the system.

These timestamps of the transactions determine the serializability order. So, if TS (T_i) < TS (T_j) then the system must ensure that the produced schedule is equivalent to a serial schedule in which T_i appears before T_j.

TIMESTAMP IMPLEMENTATION

Problems with Timestamps :

1. Each value stored in the database requires two additional timestamp fields, one for the last time the field was read and one for the last update.
2. It increases the memory requirements and the processing overhead of the database.

Implementation : To implement this scheme, we associate with each data item Q two timestamp values :

- **W-timestamp (Q)** denotes the largest timestamp of any transaction that executed write (Q) successfully.
- **R-timestamp (Q)** denotes the largest timestamp of any transaction that executed read (Q) successfully.

These timestamps are updated whenever a new read (Q) or write (Q) instruction is executed.

TIMESTAMP ORDERING PROTOCOL

- **Suppose that transaction Ti issues read (Q)** :
  - If TS(Ti)<W-timestamp (Q), then Ti needs to read a value of Q that was already overwritten. Hence the read operation is rejected, and Ti is rolled back.
  - If TS(Ti)> = W-timestamp (Q), then the read operation is executed, and R-timestamp (Q) is set to the maximum of R-timestamp(Q) i.e., TS(Ti).
- **Suppose that transaction Ti issues write (Q) :**
  - If TS(Ti)< R-timestamp (Q), then the value of Q that Ti is producing was needed previously, and the system assumed that value would never be produced. Hence, the write operation is rejected, and Ti is rolled back.
  - If TS(Ti) < W-timestamp (Q), then Ti is attempting to write an obsolete value of Q. Hence this write operation is rejected and Ti is rolled back.
  - Otherwise, the write operation is executed and W-timestamp (Q) is set to TS (Ti)

REPLICATION WITH WEAK CONSISTENCY

- Many commercial databases support replication of data with weak degrees of consistency (I.e., without a guarantee of serializability)
- E.g.: **master-slave replication**: updates are performed at a single "master" site, and propagated to "slave" sites.
  - Propagation is not part of the update transaction: its is decoupled
    - May be immediately after transaction commits
    - May be periodic
  - Data may only be read at slave sites, not updated
    - No need to obtain locks at any remote site

REPLICATION WITH WEAK CONSISTENCY (CONT.)

- Replicas should see a **transaction-consistent snapshot** of the database
  - That is, a state of the database reflecting all effects of all transactions up to some point in the serialization order, and no effects of any later transactions.
- E.g. Oracle provides a **create snapshot** statement to create a snapshot of a relation or a set of relations at a remote site
  - snapshot refresh either by re-computation or by incremental update
  - Automatic refresh (continuous or periodic) or manual refresh

MULTI-MASTER AND LAZY REPLICATION

- With multi-master replication (also called update-anywhere replication) updates are permitted at any replica, and are automatically propagated to all replicas
  - Basic model in distributed databases, where transactions are unaware of the details of replication, and database system propagates updates as part of the same transaction
    - Coupled with 2 phase commit
- Many systems support **lazy propagation** where updates are transmitted after transaction commits
  - Allows updates to occur even if some sites are disconnected from the network, but at the cost of consistency