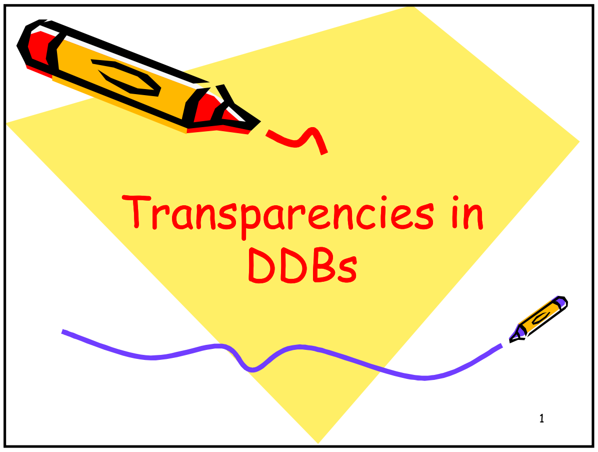
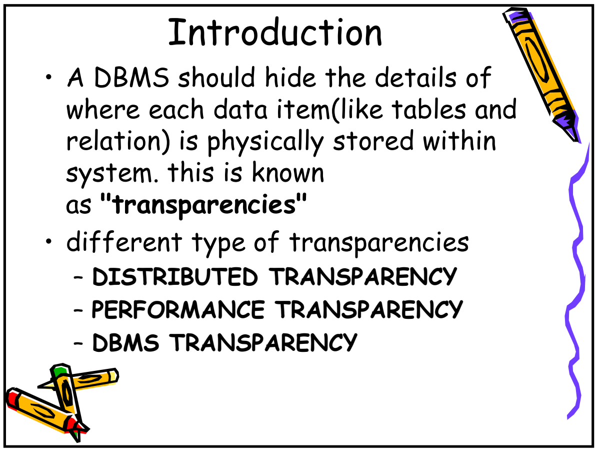
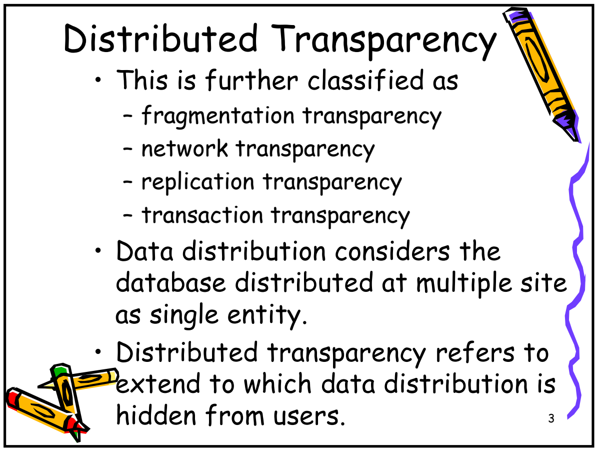


Transparencies in DDBs



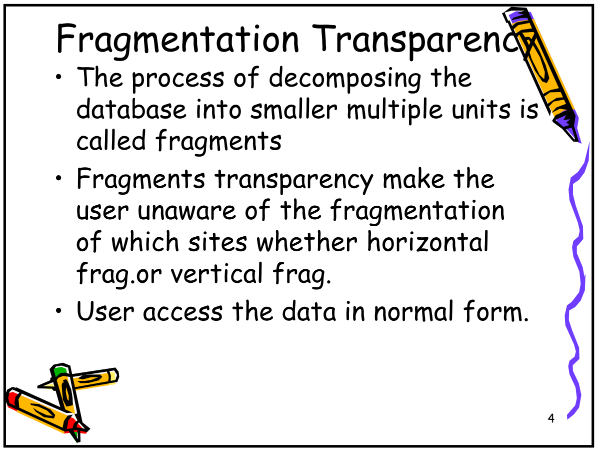
Introduction

- A DBMS should hide the details of where each data item(like tables and relation) is physically stored within system. this is known as **"transparencies"**
- different type of transparencies
  - **DISTRIBUTED TRANSPARENCY**
  - **PERFORMANCE TRANSPARENCY**
  - **DBMS TRANSPARENCY**



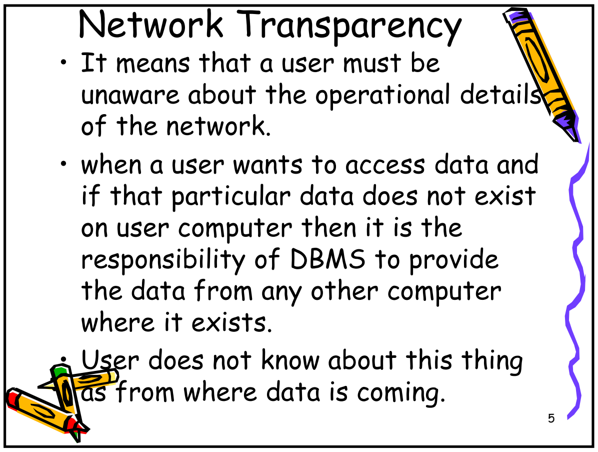
Distributed Transparency

- This is further classified as
  - fragmentation transparency
  - network transparency
  - replication transparency
  - transaction transparency
- Data distribution considers the database distributed at multiple site as single entity.
- Distributed transparency refers to extend to which data distribution is hidden from users.



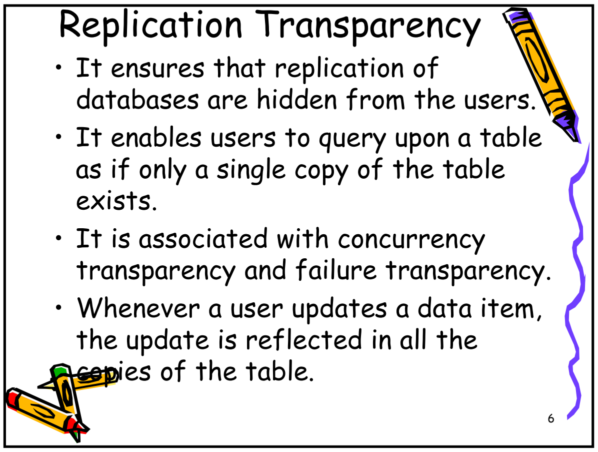
Fragmentation Transparency

- The process of decomposing the database into smaller multiple units is called fragments
- Fragments transparency make the user unaware of the fragmentation of which sites whether horizontal frag.or vertical frag.
- User access the data in normal form.



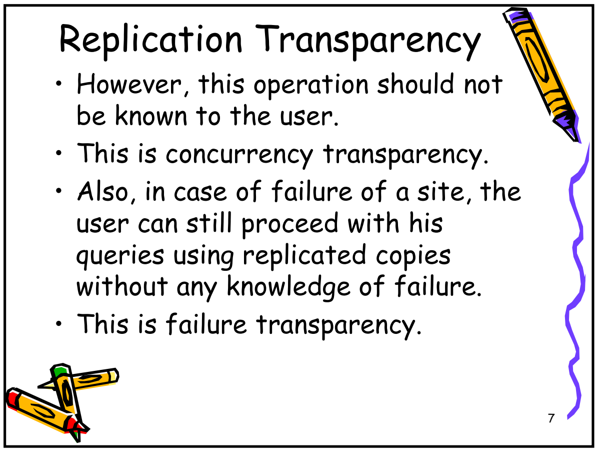
Network Transparency

- It means that a user must be unaware about the operational details of the network.
- when a user wants to access data and if that particular data does not exist on user computer then it is the responsibility of DBMS to provide the data from any other computer where it exists.
- User does not know about this thing as from where data is coming.



Replication Transparency

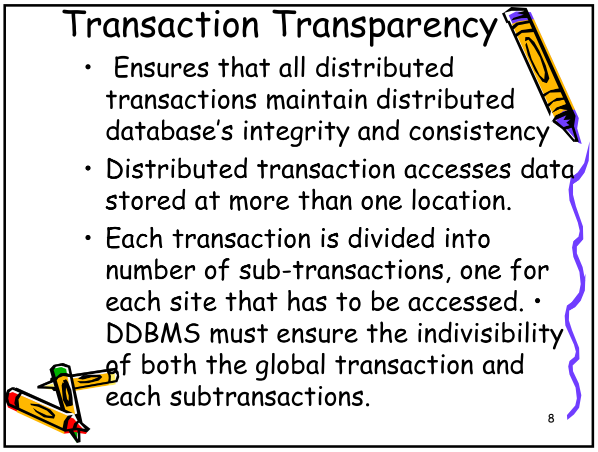
- It ensures that replication of databases are hidden from the users.
- It enables users to query upon a table as if only a single copy of the table exists.
- It is associated with concurrency transparency and failure transparency.
- Whenever a user updates a data item, the update is reflected in all the copies of the table.



Replication Transparency

- However, this operation should not be known to the user.
- This is concurrency transparency.
- Also, in case of failure of a site, the user can still proceed with his queries using replicated copies without any knowledge of failure.
- This is failure transparency.

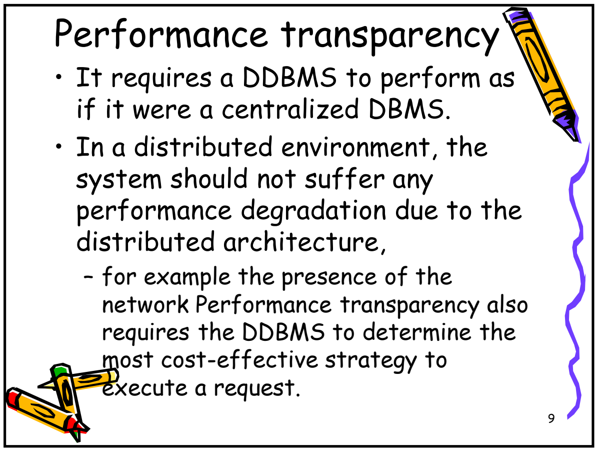
7



Transaction Transparency

- Ensures that all distributed transactions maintain distributed database's integrity and consistency
- Distributed transaction accesses data stored at more than one location.
- Each transaction is divided into number of sub-transactions, one for each site that has to be accessed. • DDBMS must ensure the indivisibility of both the global transaction and each subtransactions.

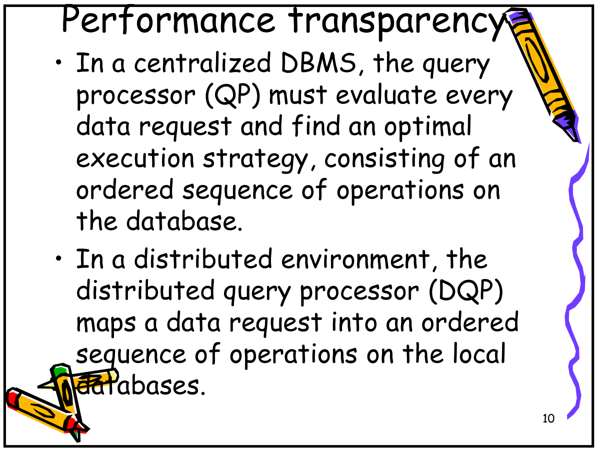
8



Performance transparency

- It requires a DDBMS to perform as if it were a centralized DBMS.
- In a distributed environment, the system should not suffer any performance degradation due to the distributed architecture,
  - for example the presence of the network Performance transparency also requires the DDBMS to determine the most cost-effective strategy to execute a request.

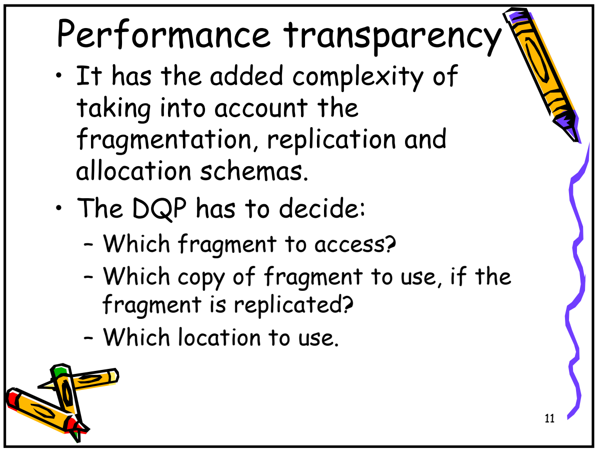
9



Performance transparency

- In a centralized DBMS, the query processor (QP) must evaluate every data request and find an optimal execution strategy, consisting of an ordered sequence of operations on the database.
- In a distributed environment, the distributed query processor (DQP) maps a data request into an ordered sequence of operations on the local databases.

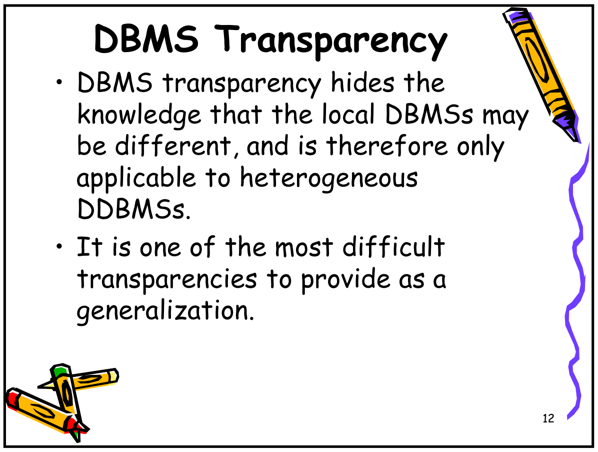
10



Performance transparency

- It has the added complexity of taking into account the fragmentation, replication and allocation schemas.
- The DQP has to decide:
  - Which fragment to access?
  - Which copy of fragment to use, if the fragment is replicated?
  - Which location to use.

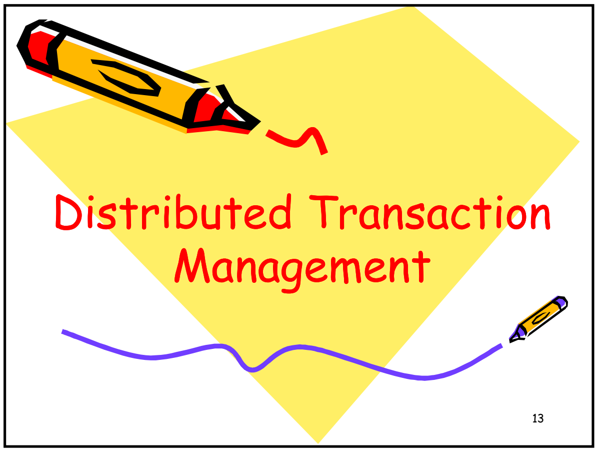
11



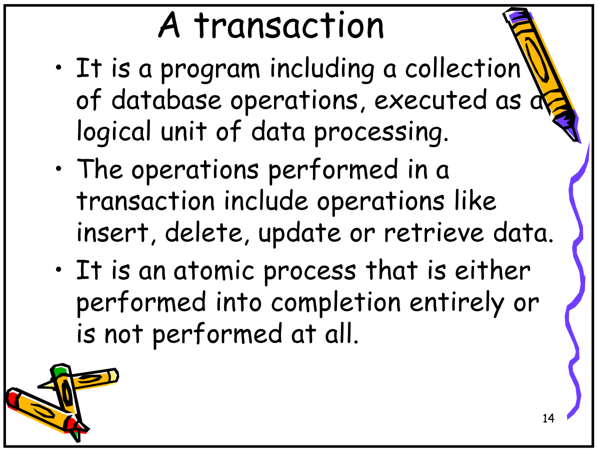
DBMS Transparency

- DBMS transparency hides the knowledge that the local DBMSs may be different, and is therefore only applicable to heterogeneous DDBMSs.
- It is one of the most difficult transparencies to provide as a generalization.

12

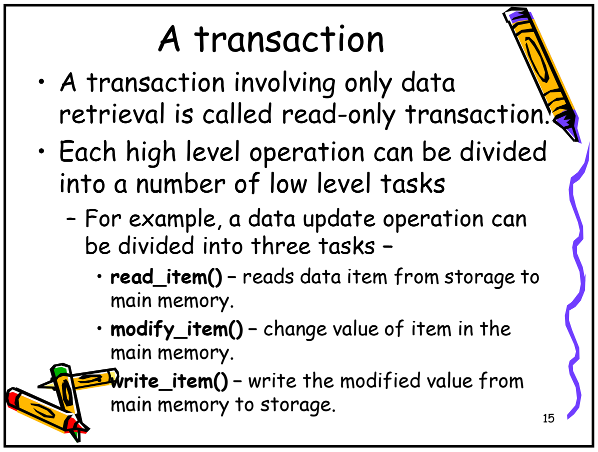


Distributed Transaction Management



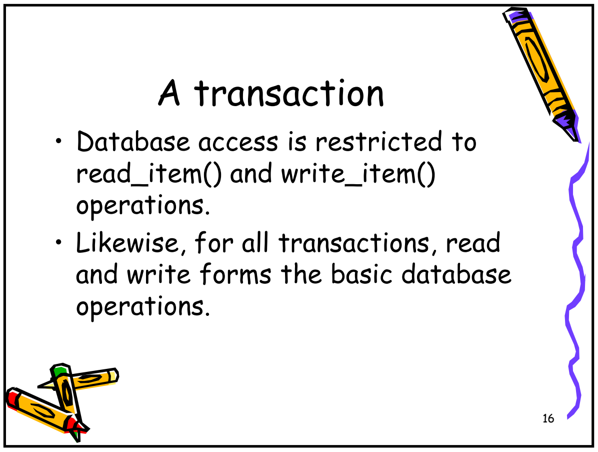
A transaction

- It is a program including a collection of database operations, executed as a logical unit of data processing.
- The operations performed in a transaction include operations like insert, delete, update or retrieve data.
- It is an atomic process that is either performed into completion entirely or is not performed at all.



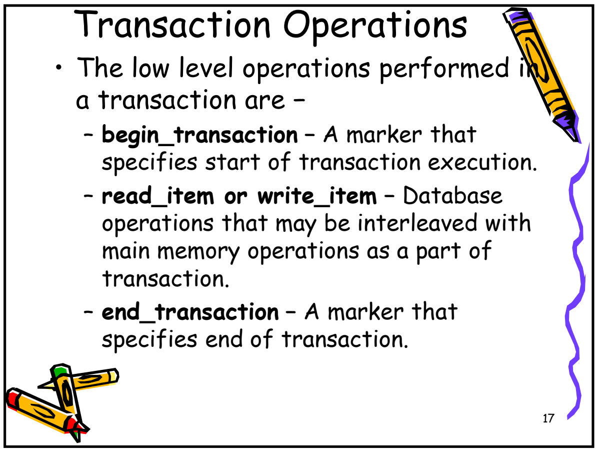
A transaction

- A transaction involving only data retrieval is called read-only transaction.
- Each high level operation can be divided into a number of low level tasks
  - For example, a data update operation can be divided into three tasks –
    - **read_item()** – reads data item from storage to main memory.
    - **modify_item()** – change value of item in the main memory.
    - **write_item()** – write the modified value from main memory to storage.



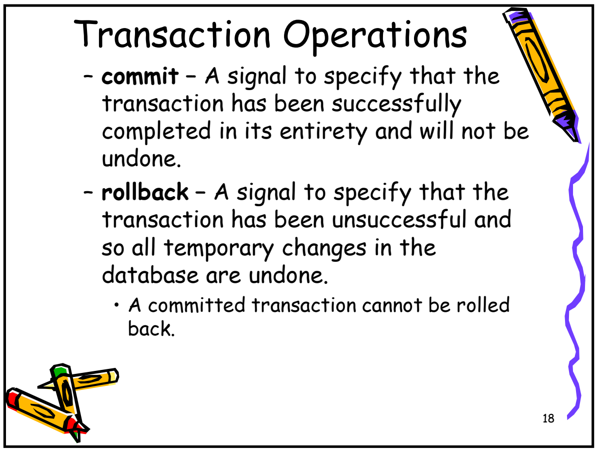
A transaction

- Database access is restricted to read_item() and write_item() operations.
- Likewise, for all transactions, read and write forms the basic database operations.



Transaction Operations

- The low level operations performed in a transaction are –
  - **begin_transaction** – A marker that specifies start of transaction execution.
  - **read_item or write_item** – Database operations that may be interleaved with main memory operations as a part of transaction.
  - **end_transaction** – A marker that specifies end of transaction.



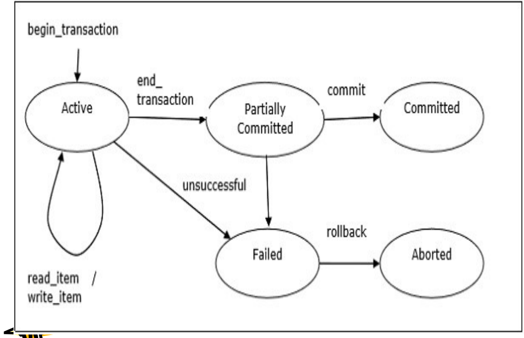
Transaction Operations

- **commit** – A signal to specify that the transaction has been successfully completed in its entirety and will not be undone.
- **rollback** – A signal to specify that the transaction has been unsuccessful and so all temporary changes in the database are undone.
  - A committed transaction cannot be rolled back.

Transaction Operations

- The following state transition diagram depicts the states in the transaction and the low level transaction operations that causes change in states.

Transaction Operations



Properties of Transactions

- Any transaction must maintain the ACID properties, viz. Atomicity, Consistency, Isolation, and Durability.
- **Atomicity** – This property states that a transaction is an atomic unit of processing, that is, either it is performed in its entirety or not performed at all.
  - No partial update should exist.

Properties of Transactions

- **Consistency** – A transaction should take the database from one consistent state to another consistent state.
  - It should not adversely affect any data item in the database.
- **Isolation** – A transaction should be executed as if it is the only one in the system.
  - There should not be any interference from the other concurrent transactions that are simultaneously running.

Properties of Transactions

- **Durability** – If a committed transaction brings about a change, that change should be durable in the database and not lost in case of any failure.

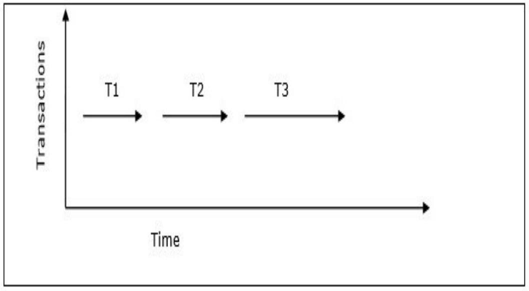
Schedules and Conflicts

- In a system with a number of simultaneous transactions, a **schedule** is the total order of execution of operations.
  - Given a schedule S comprising of n transactions, say T1, T2, T3……….Tn; for any transaction Ti, the operations in Ti must execute as laid down in the schedule S.

Types of Schedules

- There are two types of schedules –
  - **Serial Schedules** – In a serial schedule, at any point of time, only one transaction is active, i.e. there is no overlapping of transactions.
  - This is depicted in the following graph –

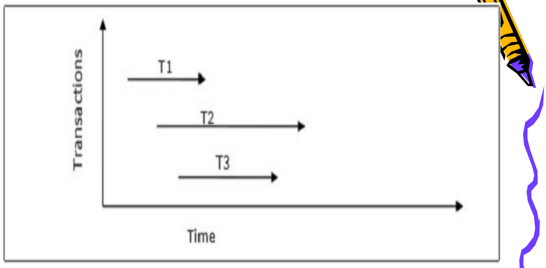
Serial Schedules



Types of Schedules

- **Parallel Schedules** – In parallel schedules, more than one transactions are active simultaneously, i.e. the transactions contain operations that overlap at time.
- This is depicted in the following graph –

Parallel Schedules

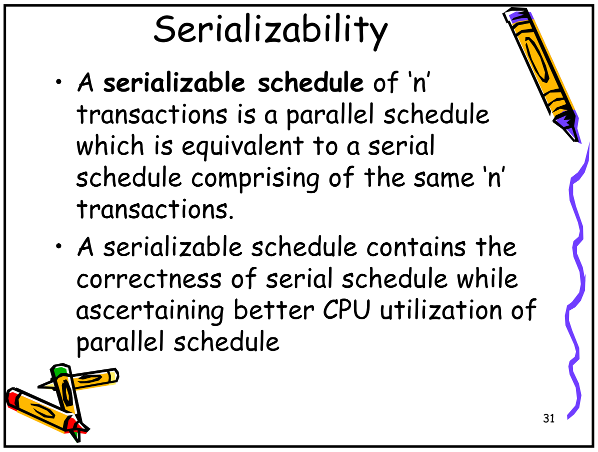


Conflicts in Schedules

- In a schedule comprising of multiple transactions, a **conflict** occurs when two active transactions perform non-compatible operations.
- Two operations are said to be in conflict, when all of the following three conditions exists simultaneously –

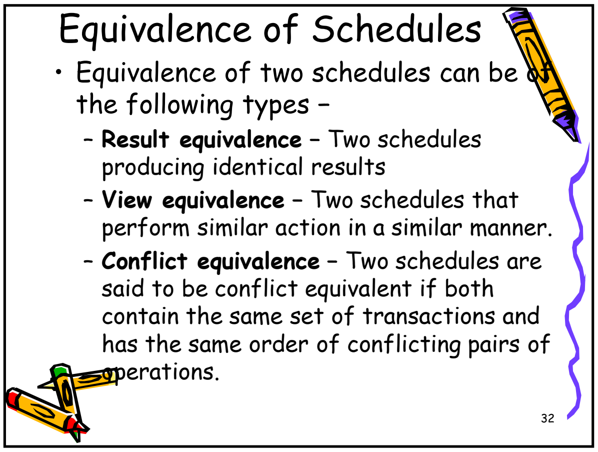
Conflicts in Schedules

- The two operations are parts of different transactions.
- Both the operations access the same data item.
- At least one of the operations is a write_item() operation, i.e. it tries to modify the data item.



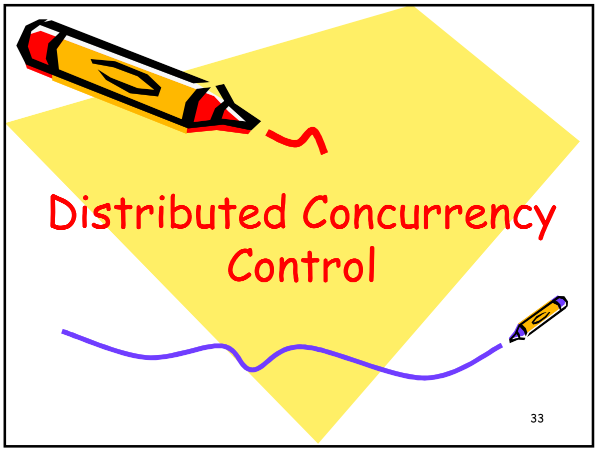
Serializability

- A **serializable schedule** of 'n' transactions is a parallel schedule which is equivalent to a serial schedule comprising of the same 'n' transactions.
- A serializable schedule contains the correctness of serial schedule while ascertaining better CPU utilization of parallel schedule

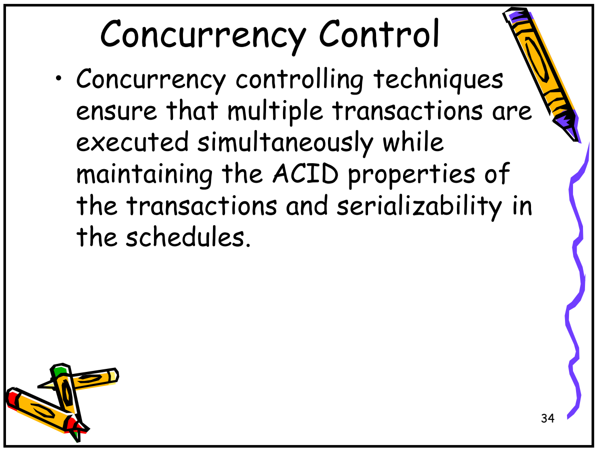


Equivalence of Schedules

- Equivalence of two schedules can be of the following types –
  - **Result equivalence** – Two schedules producing identical results
  - **View equivalence** – Two schedules that perform similar action in a similar manner.
  - **Conflict equivalence** – Two schedules are said to be conflict equivalent if both contain the same set of transactions and has the same order of conflicting pairs of operations.

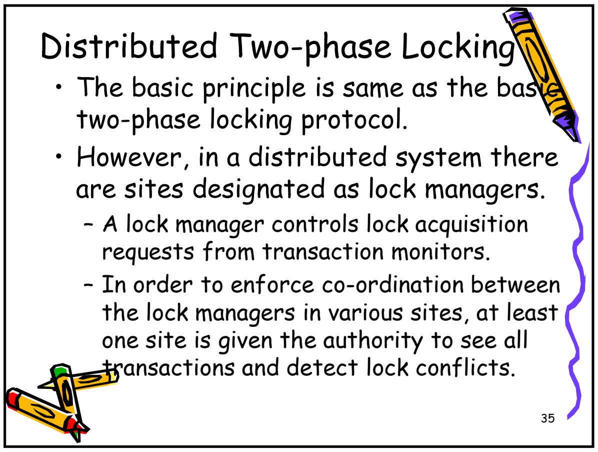


Distributed Concurrency Control



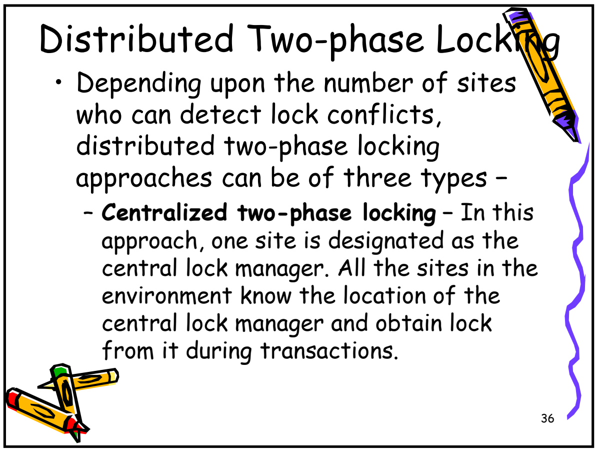
Concurrency Control

- Concurrency controlling techniques ensure that multiple transactions are executed simultaneously while maintaining the ACID properties of the transactions and serializability in the schedules.



Distributed Two-phase Locking

- The basic principle is same as the basic two-phase locking protocol.
- However, in a distributed system there are sites designated as lock managers.
  - A lock manager controls lock acquisition requests from transaction monitors.
  - In order to enforce co-ordination between the lock managers in various sites, at least one site is given the authority to see all transactions and detect lock conflicts.



Distributed Two-phase Locking

- Depending upon the number of sites who can detect lock conflicts, distributed two-phase locking approaches can be of three types –
  - **Centralized two-phase locking** – In this approach, one site is designated as the central lock manager. All the sites in the environment know the location of the central lock manager and obtain lock from it during transactions.

Distributed Two-phase Locking

- **Primary copy two-phase locking** – In this approach, a number of sites are designated as lock control centers.
- Each of these sites has the responsibility of managing a defined set of locks.
- All the sites know which lock control center is responsible for managing lock of which data table/fragment item.

Distributed Two-phase Locking

- **Distributed two-phase locking** – In this approach, there are a number of lock managers, where each lock manager controls locks of data items stored at its local site.
- The location of the lock manager is based upon data distribution and replication.

Distributed Timestamp

- In a centralized system, timestamp of any transaction is determined by the physical clock reading.
- But, in a distributed system, any site's local physical/logical clock readings cannot be used as global timestamps, since they are not globally unique.
- So, a timestamp comprises of a combination of site ID and that site's clock reading.

Distributed Timestamp

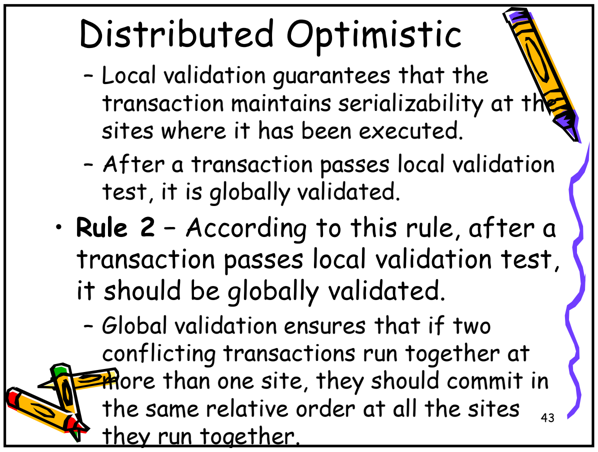
- For implementing timestamp ordering algorithms, each site has a scheduler that maintains a separate queue for each transaction manager.
- During transaction, a transaction manager sends a lock request to the site's scheduler.
- The scheduler puts the request to the corresponding queue in increasing timestamp order.

Distributed Timestamp

- Requests are processed from the front of the queues in the order of their timestamps, i.e. the oldest first.

Distributed Optimistic

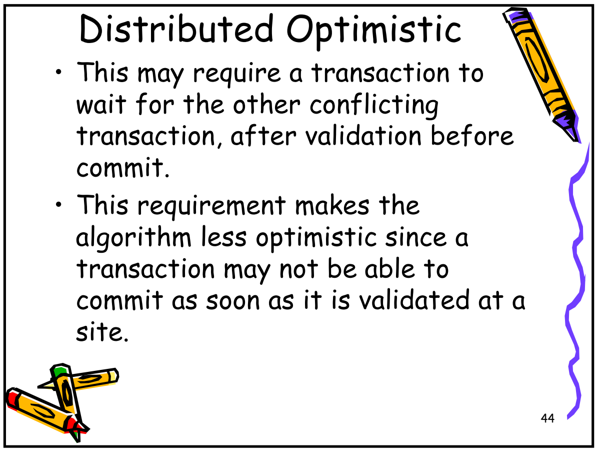
- Distributed optimistic concurrency control algorithm extends optimistic concurrency control algorithm.
- For this extension, two rules are applied
  - **Rule 1** – According to this rule, a transaction must be validated locally at all sites when it executes.
  - If a transaction is found to be invalid at any site, it is aborted.



Distributed Optimistic

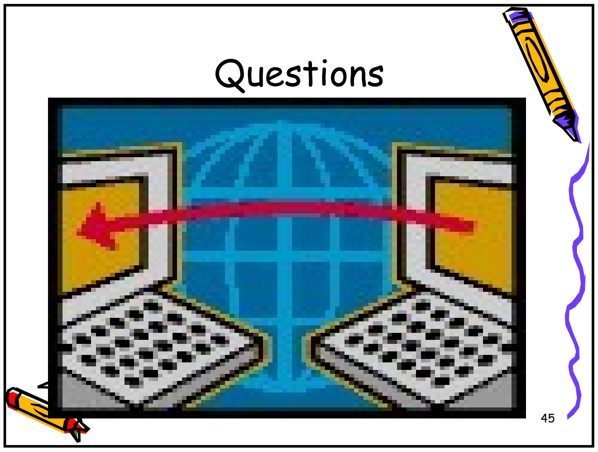
- Local validation guarantees that the transaction maintains serializability at the sites where it has been executed.
- After a transaction passes local validation test, it is globally validated.

- **Rule 2** – According to this rule, after a transaction passes local validation test, it should be globally validated.
  - Global validation ensures that if two conflicting transactions run together at more than one site, they should commit in the same relative order at all the sites they run together.



Distributed Optimistic

- This may require a transaction to wait for the other conflicting transaction, after validation before commit.
- This requirement makes the algorithm less optimistic since a transaction may not be able to commit as soon as it is validated at a site.



Questions