Mobile and Heterogeneous databases
Distributed Database System
Transaction Management

A.R. Hurson
Computer Science
Missouri Science & Technology
Distributed Databases

- Module 2 concentrated on query processing, query optimization, and query processing in distributed databases. This module will concentrate on transaction processing in general and transaction processing in distributed system. Note that we will distinguish transaction from query. A query does not change the data in base sources (e.g., relations), however, a transaction my do so. As a result, queries, initiated by several users, do not have conflicts with each other and can be executed in any order (including simultaneously). However, this is not true for transactions, since they may be in conflict with each other and hence needs to be executed in a proper sequence.
Distributed Databases

- **Summary Last section:**
  - Decomposition of Global query into sub-queries (local queries) (use of catalog, meta data, …),
  - Execution plan at the global level (reduce the communication cost) – data distribution, global execution of operations, data integration, computational and storage capability of local nodes, …
  - Execution plan at each node (reducing the I/O operations, and reducing the execution time and storage requirements) – Underlying architecture at each node.
In this module, we will talk about:

- Transaction processing and management
- Formal definition of transactions
- ACID property
- Serializability
- Concurrency control
- Concurrency control protocols
- Transaction processing
- Transaction processing in distributed system
Distributed Databases

- Distributed transaction management
  - Two potential problems:
    - Two requests attempt to update the same data item, simultaneously, and
    - system fails during the execution of a request.
  - In case of retrieve only request (e.g., query processing):
    - The 1st issue has no consequences, and
    - The 2nd issue is resolved by restarting the request.
In the following few slides, we will define several terms that should motivate:

- Issues of concern in transaction processing,
- Capability of the database system in transaction processing, and
- Characteristics of a transaction.
Distributed Databases

- **Distributed transaction management**
  - In a query processing there is no notion of consistent execution or reliable execution. These issues are becoming a part of transaction processing.
  - A *transaction* is a basic unit of consistent and reliable computing.
  - We distinguish a difference between database consistency and transaction consistency.
Distributed Databases

- **Distributed transaction management**
  - A database is in consistent state if it obeys all integrity constraints defined over it.
  - State of a database changes due to the update operations—modifications, insertions, and deletions.
  - Database can be temporarily inconsistent during the execution of a transaction. The important point is that the database should be in consistent state when the transaction terminates.
  - Transaction consistency refers to the actions of concurrent transactions—we would like database remain in a consistent state even if there are a number of concurrent users’ transactions.
Distributed Databases

- **Distributed transaction management**
- A transaction is a sequence of operations that transfers database from one **consistent state** to another **consistent state**.

Start of transaction

Database in Consistent state

Execution of transaction
Database might be temporarily in inconsistent state.

End of transaction

Database in Consistent state
Distributed Databases

- Distributed transaction management
  - Several issues hinder transaction consistency:
    - Concurrent execution of transactions,
    - Replicated data, and
    - Failure.
  - A replicated database is in a mutually consistent state if copies of every data item in it have identical values — one copy equivalence.
Distributed Databases

- **Distributed transaction management**
  - **Reliability** refers to **resiliency** of a system to various types of failures and its ability to **recover** from it.
    - A **resilient system** tolerates system failure and continue to provide services,
    - A **recoverable system** is the one that can get to a consistent state under failure.
Distributed Databases

- Distributed transaction management — Example

- Assume the following database (airline reservation system):

```
<table>
<thead>
<tr>
<th>Flight no</th>
<th>Source</th>
<th>Destination</th>
<th>Seats sold</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNO</td>
<td>DATE</td>
<td>SRC</td>
<td>DEST</td>
<td>STSOLD</td>
</tr>
</tbody>
</table>
```

Flight relation

Capacity

Source

Seats sold
Distributed Databases

- Distributed transaction management — Example

Customer relation

FC relation

<table>
<thead>
<tr>
<th>CNAME</th>
<th>ADDR</th>
<th>BAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FNO</th>
<th>DATE</th>
<th>CNAME</th>
<th>SPECIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Customer Address

Account Balance

Flight no

Special request
Distributed Databases

- **Distributed transaction management — Example**

  
  Begin_transaction Reservation
  Begin
  
  input (flight-no, date, customer-name)
  EXEC SQL UPDATE FLIGHT
  SET STSOLD = STSOLD + 1
  WHERE FNO = flight-no
  AND DATE = date
  EXEC SQL INSERT
  INTO FC(FNO,DATE,CNAME,SPECIAL)
  VALUES (flight-no,date,customer-name,null);
  
  output (“reservation completed”)
  
  end
Distributed Databases

- **Distributed transaction management — Example**
  - The previous example assumed that there will always be a free seat available. However, transaction might fail because the plane is full. A transaction must always terminate even if there is a failure.
  - If a transaction completes its task successfully, then the transaction must commit — its results will be available to other transactions.
  - If a transaction stops without completing its task then it must be aborted — all its already performed operations must be undone.
Distributed transaction management — Example

```
Begin_transaction Reservation

Begin

Input (flight-no, date, customer-name)
EXEC SQL SELECT STSOLD,CAP
    INTO temp1,temp2
    FROM FLIGHT
    WHERE FNO = flight-no
    AND DATE = date;

if temp1 = temp2 then
begin
    output (“no available seats”);
    Abort
end
else begin

EXEC SQL UPDATE FLIGHT
    SET STSOLD = STSOLD + 1
    WHERE FNO = flight-no
    AND DATE = date;
EXEC SQL INSERT
    INTO FC(FNO,DATE,CNAME,SPECIAL)
    VALUES (flight-no,date,customer-name,null);
    Commit;
    output (“reservation completed”)
end

end-if
end
```
Distributed Databases

Let us define a set of notations that allows us to formally represent a transaction:
Distributed Databases

- **Distributed transaction management**
  - The data items read by a transaction is called **read set (RS)**.
  - The data item that a transaction writes are called **write set (WS)**.
  - The base set for a transaction is defined as:
    \[ BS = RS \cup WS \]
Distributed Databases

- Distributed transaction management

- In our previous example:
  \[ \text{RS} = \{ \text{STSOLD, CAP} \} \]
  \[ \text{WS} = \{ \text{STSOLD, FNO, DATE, CNAME, SPECIAL} \} \]
  \[ \text{BS} = \{ \text{STSOLD, CAP, FNO, DATE, CNAME, SPECIAL} \} \]
Distributed Databases

- Distributed transaction management

For a transaction $T_i$:

$O_{ij}(x) \in T_i$ (operation $O_j$ of transaction $T_i$ operating on data item $x$)

$O_{ij}(x) \in \{\text{read, write}\}$ (operations are atomic)

$OS_i = \bigcup_j O_{ij}$

$N_i \in \{\text{abort, commit}\}$ ($N_i$ the termination condition)

$T_i = \{\Sigma_i, \preceq_i\}$

$\Sigma_i = OS_i \cup \{N_i\}$

$\preceq$ is a binary operator representing the execution order

$O_{ij}, O_{ik} \in OS_i$, if $O_{ij} = \{R(x) \text{ or } W(x)\}$ and $O_{ik} = W(x)$ for any data item $x$, then either $O_{ij} \preceq_i O_{ik}$ or $O_{ik} \preceq_i O_{ij}$

$\forall O_{ij} \in OS_i, O_{ij} \preceq_i N_i$
Distributed Databases

- Distributed transaction management

Consider the following transaction, its formal definition, and its graphical representation (T):

\[ \Sigma = \{ \text{R}(x), \text{R}(y), \text{W}(x), \text{C}\} \]
\[ \prec = \{ (\text{R}(x), \text{W}(x)), (\text{R}(y), \text{W}(x)), (\text{W}(x), \text{C}), (\text{R}(x), \text{C}), (\text{R}(y), \text{C})\} \]

Where \((O_i, O_j)\) as an element indicates that \(O_i \prec O_j\)
Distributed Databases

- Distributed transaction management
  - As another example, recall our earlier reservation transaction. Also remember that the reservation transaction had two terminating conditions.
  - First part can be formally defined as:

\[
\Sigma = \{ R(STSOLD), R(CAP), A \}
\]

\[
\prec = \{ (O_1, A), (O_2, A) \}
\]
Distributed Databases

- Distributed transaction management
- The second part can be represented as:

\[ \Sigma = \{ \text{R(STSOLD), R(CAP), W(STSOLD), W(FNO), W(DATE), W(CNAME), W(SPECIAL), C} \} \]

\[ \prec = \{ \text{(O}_1\text{, O}_3\text{), (O}_2\text{, O}_3\text{), (O}_1\text{, O}_4\text{), (O}_1\text{, O}_5\text{), (O}_1\text{, O}_6\text{), (O}_1\text{, O}_7\text{), (O}_2\text{, O}_4\text{), (O}_2\text{, O}_5\text{), (O}_2\text{, O}_6\text{), (O}_2\text{, O}_7\text{), (O}_1\text{, C), (O}_2\text{, C), (O}_3\text{, C), (O}_4\text{, C), (O}_5\text{, C), (O}_6\text{, C), (O}_7\text{, C)} \} \]

Where \( O_1 = \text{R(STSOLD)} \), \( O_2 = \text{R(CAP)} \), \( O_3 = \text{W(STSOLD)} \), \( O_4 = \text{W(FNO)} \), \( O_5 = \text{W(DATE)} \), \( O_6 = \text{W(CNAME)} \), and \( O_7 = \text{W(SPECIAL)} \)
Distributed Databases

- Distributed transaction management
  - In general, in a database system, one needs to ensure Atomicity, Consistency, Isolation, and Durability properties of transactions:
Distributed Databases

- **Distributed transaction management**
  - **Atomicity** (all or nothing): either all operations of the transaction are reflected in database, or none are.
  - **Consistency** (no violation of integrity rules): Execution of transaction in isolation preserves the consistency of the database.
  - **Isolation** (Concurrent changes invisible and serializable): Even though multiple transactions may execute concurrently, each transaction assumes it is executed in isolation (it is unaware of other transactions executing concurrently in the system).
  - **Durability** (Committed updates persist): After a transaction completes successfully, its results are becoming persistence.
Distributed Databases

- Distributed transaction management — Atomicity

- The database should always reflect a real state of the world.

- A transaction must transfer the database from one consistent state to another.

- If during the course of a transaction a failure occurs, then the database is in inconsistent state and it does not reflect a real world state. Therefore, the partial results must be undone.
Distributed Databases

- **Distributed transaction management** — Atomicity
  - The activity of preserving the transaction’s atomicity in the presence of aborts due to input data errors, system overheads, or deadlock is called *transaction recovery*.
  - The activity of ensuring atomicity in the presence of system crashes is called *crash recovery*.
Distributed Databases

- **Distributed transaction management** — Consistency
  - If the database is consistent before execution of a transaction, the database remains consistent after the execution of the transaction.
  - Transactions are correct programs that do not violate database integrity constraints.
Distributed Databases

- Distributed transaction management — Consistency

- From consistency point of view four levels of consistency can be recognized:
  - Degree 3: a transaction sees degree 3 consistency if:
    - $T$ does not overwrite dirty data of other transactions (preventing lost update),
    - $T$ does not commit any writes until it completes all its writes — until the end of transaction,
    - $T$ does not read dirty data from other transactions,
    - Other transactions do not dirty any data read by $T$ before $T$ completes.
Distributed Databases

- Distributed transaction management
- Consistency

- Degree 2: a transaction sees degree 2 consistency if:
  - $T$ does not overwrite dirty data of other transactions,
  - $T$ does not commit any writes until it completes all its writes — until the end of transaction,
  - $T$ does not read dirty data from other transactions.
Distributed Databases

- **Distributed transaction management** — Consistency
  - Degree 1: a transaction sees degree 1 consistency if:
    - $T$ does not overwrite dirty data of other transactions,
    - $T$ does not commit any writes until it completes all its writes — until the end of transaction.
  - Degree 0: a transaction sees degree 0 consistency if:
    - $T$ does not overwrite dirty data of other transactions.

**Dirty read**: Data item whose value is modified by an un-committed transaction
Distributed Databases

- **Distributed transaction management** — Isolation

- To improve performance, we need to **interleave operations** of transactions running concurrently.

- Even if **consistency** and **atomicity** properties are ensured, undesirable interleaving of operations results in an inconsistent state.

- Isolation property, guarantees that concurrent transactions are interleaved correctly.
Distributed Databases

- Distributed transaction management — Isolation
  - Serilizability: If several transactions are executed concurrently, the result must be the same as if they were executed serially in an orderly fashion.
  
  - Incomplete results: Result of an incomplete transactions is not available to other transactions before it is committed.
  
  - Cascading aborts: In execution of concurrent transactions, attempts must be made to avoid cascading aborts. Cascading aborts happens if a transaction allows other transactions to see its incomplete result before committing and later on deciding to abort.
Distributed Databases

- Distributed transaction management — Isolation
- Consider the following two transactions and their possible scheduling orders:

\[ \begin{align*}
T_1: & \quad \text{Read (x)} \\
& \quad x \leftarrow x + 1 \\
& \quad \text{Write (x)} \\
& \quad \text{Commit} \\
T_2: & \quad \text{Read (x)} \\
& \quad x \leftarrow x + 1 \\
& \quad \text{Write (x)} \\
& \quad \text{Commit}
\end{align*} \]
Distributed Databases

- Distributed transaction management
  - Isolation

Correct execution order:

- $T_1$: Read (x)
- $T_1$: x $\leftarrow$ x + 1
- $T_1$: Write (x)
- $T_1$: Commit
- $T_2$: Read (x)
- $T_2$: x $\leftarrow$ x + 1
- $T_2$: Write (x)
- $T_2$: Commit

Incorrect execution order:

- $T_1$: Read (x)
- $T_1$: x $\leftarrow$ x + 1
- $T_2$: Read (x)
- $T_1$: Write (x)
- $T_2$: x $\leftarrow$ x + 1
- $T_2$: Write (x)
- $T_1$: Commit
- $T_2$: Commit
Distributed Databases

- Distributed transaction management — Durability
  - After successful termination of a transaction, no system failure should result in a loss of data.
  - The durability properly guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist.
Distributed Databases

- Summary Last lecture
  - Transaction processing vs. Query processing
  - Formal definition of transaction
  - ACID property
  - Consistency levels
Distributed Databases

- **Distributed transaction management — Classification**
  - Based on different parameters, different types of transactions can be recognized:
    - **Structure**: flat transaction vs. nested transactions.
    - **Timing (duration)**: short life (on-line) transactions vs. long life (batch) transactions, conversational transactions.
    - **Application areas**: centralized transactions vs. distributed transactions.
    - **Organization** of read and write actions.
Distributed Databases

- Distributed transaction management — Classification
  - Flat Transaction: It is a sequence of primitive operations (read, write, commit).
  - Nested transaction: The operations of the transaction may themselves be transactions.
Distributed Databases

- Distributed transaction management — Flat Transaction

  `Begin_transaction Reservation`

- 
- 
- 

  `end`
Distributed Databases

- Distributed transaction management
  - Nested transaction
    - Begin_transaction Reservation
    - Begin_transaction Airline
    - end (airline)
    - Begin_transaction Hotel
    - end (hotel)
    - end (Reservation)
Distributed Databases

- Distributed transaction management — Classification
  - Closed nesting
    - Sub-transactions begin after their parents and finish before the parents. Commitment of a sub-transaction is conditional upon the commitment of the parent.
  - Open nesting
    - Sub-transactions can execute and commit independently. In case of open nesting we may be needing compensating transaction.
Distributed Databases

- **Distributed transaction management** — Classification
  - **Two-step transaction**: All read actions are performed before write actions.
  - **Restricted**: A data item has to be read before being updated.
  - **Restricted two-step**: A transaction that is both two-step and restricted.
  - **Action Model**: A restricted model with additional restriction that each <read, write> pair be executed atomically.
Distributed Databases

- Distributed transaction management — Classification

General Transaction
\[ T = \{ \text{R}(x), \text{R}(y), \text{W}(y), \text{R}(z), \text{W}(x), \text{W}(z), \text{W}(w), \text{C} \} \]

Two-step Transaction
\[ T = \{ \text{R}(x), \text{R}(y), \text{R}(z), \text{W}(x), \text{W}(z), \text{W}(y), \text{W}(w), \text{C} \} \]

Restricted Transaction
\[ T = \{ \text{R}(x), \text{R}(y), \text{W}(y), \text{R}(z), \text{W}(x), \text{W}(z), \text{R}(w), \text{W}(w), \text{C} \} \]

Restricted Two-step Transaction
\[ T = \{ \text{R}(x), \text{R}(y), \text{R}(z), \text{R}(w), \text{W}(x), \text{W}(z), \text{W}(y), \text{W}(w), \text{C} \} \]

Action Transaction
\[ T = \{ [\text{R}(x), \text{W}(x)], [\text{R}(y), \text{W}(y)], [\text{R}(z), \text{W}(z)], [\text{R}(w), \text{W}(w)], \text{C} \} \]
Distributed Databases

- **Distributed transaction management — Classification**

![Diagram]

- General Model
  - Two-step Model
  - Restricted Two-step Model
  - Restricted Model
  - Action Model
Distributed Databases

- Summary
  - Definition of transaction
  - ACID property
  - Levels of consistency
  - Cascading aborts
  - Different classes of transactions
Distributed Databases

- **Distributed transaction management** – Transaction states

  - In the absence of failures, we are expecting that a transaction completes successfully.
  - A transaction that completes its execution successfully is said to be **committed**.
  - A committed transaction, that has updated the database, has transferred database from one consistent state to a new consistent state which must be persisted, even if the system fails (database recovery).
Distributed Databases

- **Distributed transaction management** —
  
  **Transaction states**

  - Note, if a transaction is committed, we cannot undo its effect by _aborting_ it — We need a _compensating transaction_ to undo its effect.

  - If a transaction does not complete its execution successfully, to ensure atomicity, it must be _aborted_ — and any change to database must be undone.

  - In case of failure, the transaction must be _rolled back_.

Distributed Databases

- Distributed transaction management

Transaction states

- In general, a transaction is in one of the following states:
  - **Active**: the transaction stays in this state while executing,
  - **Partially committed**: the final statement of transaction has been executed,
  - **Failed**: it is discovered that normal execution can no longer proceed,
  - **Aborted**: the transaction has been rolled back and state of database has been restored.
  - **Committed**: successful completion of transaction.
Distributed Databases

- Distributed transaction management — Transaction states

Distributed transaction management

- Active
- Partially committed
- Failed
- Committed
- Aborted
Distributed Databases

- **Distributed transaction management**
  - It is much easier if internally consistent transactions are run *serially* — each transaction is executed alone, one after the another. However, there are two good motivations to allow concurrent execution of transactions:
    - Improved *throughput* and *resource utilization*
    - Improved *average response time*.
  - Concurrent execution of transactions means that they should be *scheduled* in order to ensure consistency.
Distributed Databases

- Distributed transaction management
  - The concurrency control mechanism attempts to find a suitable trade-off between maintaining the consistency of the database and maintaining a high level of concurrency.
  - Note concurrency control deals with the isolation and consistency properties of transactions.
Distributed Databases

- Distributed transaction management
  - Two operations (within a transaction or two transactions) are in conflict if their order of execution is important:
    - Read-write,
    - Write-read,
    - Write-write.
Distributed Databases

- Distributed transaction management
  - A schedule (a history) over a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ is an interleaved order of execution of these transactions.
  - A schedule is a complete schedule, if it defines the execution order of all operations in its domain.
Distributed Databases

**Distributed transaction management**

Formally a complete schedule $S_T^C$ over a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ is a partial order $S_T^C = \{\Sigma_T, \prec_T\}$ where:

$$\Sigma_T = \bigcup_{i=1}^{n} \Sigma_i$$

$$\prec_T \supseteq \bigcup_{i=1}^{n} \prec_i$$

For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$ either $O_{ij} \prec_T O_{kl}$ or $O_{kl} \prec_T O_{ij}$.

1\textsuperscript{st} rule shows that the schedule must contain all operations in participating transactions.

2\textsuperscript{nd} rule shows that the ordering relation on $T$ is a superset of ordering relations of individual transactions.

3\textsuperscript{rd} rule shows the execution order among conflicting operations.
Distributed Databases

- Distributed transaction management
  - Consider the following two transactions:

  \[
  \begin{align*}
  S_T^C & = \{ \Sigma_r, \prec_r \} \\
  T_1: & \quad \text{Read } (x) \quad \text{Write } (x) \\
           & \quad x \leftarrow x + 1 \\
           & \quad \text{Commit} \\
  T_2: & \quad \text{Read } (x) \quad \text{Write } (x) \\
           & \quad x \leftarrow x + 1 \\
           & \quad \text{Commit}
  \end{align*}
  \]
Distributed Databases

- Distributed transaction management

\[ \Sigma_1 = \{ R_1(x), W_1(x), C_1 \} \]
\[ \Sigma_2 = \{ R_2(x), W_2(x), C_2 \} \]
\[ \Sigma_T = \{ R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2 \} \]
\[ \prec_T = \{ (R_1, R_2), (R_1, W_1), (R_1, C_1), (R_1, W_2), (R_1, C_2), \]
\[ (R_2, W_1), (R_2, C_1), (R_2, W_2), (R_2, C_2), (W_1, C_1), \]
\[ (W_1, W_2), (W_1, C_2), (C_1, W_2), (W_2, C_2), (C_1, C_2) \} \]

Transitive relationships are omitted for the sake of clarity.
Distributed Databases

Question

Consider the following transactions:

- Define its complete schedule and its corresponding DAG.

What is an action model transaction?
Distributed Databases

- Distributed transaction management
- Consider the following transactions:

$T_1$: Read (A);  
A := A – 50;  
Write (A);  
Read (B);  
B := B + 50;  
Write (B);

$T_2$: Read (A);  
temp := A * 0.1;  
A := A – temp;  
Write (A);  
Read (B);  
B := B + temp;  
Write (B);
Distributed Databases

- Distributed transaction management

- The following is the serial execution schedule of $T_1$ followed by $T_2$:

```
Read (A);
A := A – 50;
Write (A);
Read (B);
B := B + 50;
Write (B);
```

```
Read (A);
temp := A * 0.1;
A := A – temp;
Write (A);
Read (B);
B := B + temp;
Write (B);
```
Distributed Databases

- Distributed transaction management
  - A schedule for a set of transactions must consist of all instructions in those transactions.
  - A serial schedule consists of a sequence of instructions in transactions, where instructions of one single transaction appear together in that schedule.
Distributed Databases

- Distributed transaction management

For the following transactions:

$T_1$:
- Read (x)
- Write (x)
- Commit

$T_2$:
- Write (x)
- Write (y)
- Read (z)
- Commit

$T_3$:
- Read (x)
- Read (y)
- Read (z)
- Commit

$S = \{W_2(x), W_2(y), R_2(z), C_2, R_1(x), W_1(x), C_1,$
$R_3(x), R_3(y), R_3(z), C_3\}$

Is a serial schedule since $T_2$ is executed before $T_1$ and $T_1$ is executed before $T_3$:

$T_2 \prec_S T_1 \prec_S T_3 \quad \rightarrow \quad T_2 \rightarrow T_1 \rightarrow T_3$
Distributed Databases

- Distributed transaction management
  - When interleaving instructions from different transactions, one can come up with a number of execution sequence (schedule).
  - In this case, we can ensure consistency if the concurrent schedule has the same effect as a serial schedule of transactions — Concurrent schedule is equivalent to a serial schedule.
Distributed Databases

- Distributed transaction management

Read (A);
A := A – 50;
Write (A);

Read (A);
temp := A * 0.1;
A := A – temp;
Write (A);

Read (B);
B := B + 50;
Write (B);

Read (B);
B := B + temp;
Write (B);
Distributed Databases

- Distributed transaction management — Conflict Serializability

- In two operations of two transactions refer to two different data items, they can be executed in any order. We might have problem if these operations refer to the same data item.
Distributed Databases

- Distributed transaction management
- Conflict Serializability

In general, assume two transactions $T_i$ and $T_j$ and two instruction $I_i \in T_i$ and $I_j \in T_j$:

- If $I_i = \text{Read}(Q)$ and $I_j = \text{Read}(Q)$, the order of $I_i$ and $I_j$ does not matter.
- If $I_i = \text{Read}(Q)$ and $I_j = \text{Write}(Q)$, the order of $I_i$ and $I_j$ matters.
- If $I_i = \text{Write}(Q)$ and $I_j = \text{Read}(Q)$, the order of $I_i$ and $I_j$ matters.
- If $I_i = \text{Write}(Q)$ and $I_j = \text{Write}(Q)$, the order of $I_i$ and $I_j$ matters.

$I_i$ and $I_j$ conflicts if they are operations of two different transactions on the same data item and at least one of them is a write operation.
Distributed Databases

- Distributed transaction management
  Conflict Serializability

- A simplified version of previous transactions.
Distributed Databases

- Distributed transaction management
  - Conflict Serializability

- Two schedules $S$ and $S'$ are conflict equivalent if $S'$ is generated by a series of swaps of non conflicting instructions in $S$. 

```
Read (A); Write (A);
Read (B); Write (B);
Read (A); Write (A);
Read (B); Write (B);
Read (A); Write (A);
Read (B); Write (B);
Read (A); Write (A);
Read (B); Write (B);
```
Distributed Databases

- Distributed transaction management – Conflict Serializability

- Formally, two schedules $S$ and $S'$ over a set of transactions are conflict equivalent if for each pair of conflicting operations $O_{ij}, O_{kl}$ ($i \neq k$), whenever $O_{ij} \prec_S O_{kl}$, then $O_{ij} \prec_{S'} O_{kl}$. 
Distributed Databases

Distributed transaction management

Conflict Serializability

Consider the following transactions:

**T₁:**
- Read (x)
- Write (x)
- Commit

**T₂:**
- Write (x)
- Write (y)
- Read (z)
- Commit

**T₃:**
- Read (x)
- Read (y)
- Read (z)
- Commit

The schedule \( S' = \{W₂(x), R₁(x), W₁(x), C₁, R₃(x), W₂(y), R₃(y), R₂(z), C₂, R₃(z), C₃\} \)

Is conflict equivalence to schedule

\[ S = \{W₂(x), W₂(y), R₂(z), C₂, R₁(x), W₁(x), C₁, R₃(x), R₃(y), R₃(z), C₃\} \]
Distributed Databases

- Distributed transaction management — Conflict Serializability
  - Concept of conflict equivalent leads to the concept of conflict serializability.
  - A schedule $S$ is conflict serializable if it is conflict equivalent to a serial schedule.
  - Note that serializability is roughly equivalent to degree 3 consistency discussed before.
Distributed Databases

Summary
- Transaction states
- Concurrency control
- Serial schedule
- Conflict equivalent
- Conflict serializable
- Hw#2
Distributed Databases

- Distributed transaction management

Uncommitted Data (Dirty read (WR conflict))

Assume $T_1$ transfers 100 from A to B, and $T_2$ increments both A and B by 6%.

The sequence of operations as scheduled does not generate the same data in A and B as the serial execution of $T_1$ and $T_2$, regardless of the order.
Distributed Databases

- Distributed transaction management
  - Unrepeatable reads (WR conflict)

$T_1$ reads two different values for A.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read (A);</td>
<td>Read (A);</td>
</tr>
<tr>
<td>Read (A);</td>
<td>Write (A);</td>
</tr>
</tbody>
</table>
Distributed Databases

- Distributed transaction management

Unrepeatable reads (WR conflict)

Assume $T_1$ increments A and $T_2$ decrements A

The sequence of operations as scheduled does not generate the same data in A as the serial execution of $T_1$ and $T_2$, regardless of the order.
Distributed Databases

- Distributed transaction management
  Overwriting Uncommitted data (WW conflict (blind write))

Assume $T_1$ and $T_2$ are intended to keep the same values in A and B. Say $T_1$ sets A and B to 2000 and $T_2$ sets A and B to 1000.

The sequence of operations as scheduled does not generate the same data in A and B as the serial execution of $T_1$ and $T_2$, regardless of the order.
Distributed transaction management

Scheduling Involving Aborted Transactions

In this case, all actions of $T_1$ are to be undone. However, $T_2$ is already committed. If $T_2$ was not committed by cascading aborts we were able to resolve the situation. Such a schedule is called **unrecoverable schedule**.
A serializable schedule over a set of T transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of committed transactions in T.
Distributed Databases

- Distributed transaction management — Testing for Serializability
  - The concept of precedence graph can be used to test serializability.
  - A precedence graph for a schedule $S$ is a directed graph $G = (V, E)$, where $V$ is the set of vertices each representing a transaction and $E$ is the set of directed edges between the vertices.
Distributed Databases

- Distributed transaction management — Testing for Serializability

Assume $T_i$ and $T_j \in V$, then there is an edge between $T_i \rightarrow T_j$ if one of the following conditions holds:

- If $T_i$ executes Read(Q) before $T_j$ executes Write(Q),
- If $T_i$ executes Write(Q) before $T_j$ executes Read(Q),
- If $T_i$ executes Write(Q) before $T_j$ executes Write(Q),
Distributed Databases

- Distributed transaction management – Testing for Serializability
  - If the precedence graph for a schedule contains a cycle, then this schedule is not conflict serializable, otherwise it is.
Distributed Databases

- Distributed transaction management —

A schedule and its precedence graph

Read (A);
A := A – 50;
Write (A);
Read (B);
B := B + 50;
Write (B);

Read (A);
temp := A * 0.1;
A := A – temp;
Write (A);
Read (B);
B := B + temp;
Write (B);
Distributed Databases

- Distributed transaction management

A schedule and its precedence graph

```
Read (A);
A := A - 50;

Write (A);
Read (B);
B := B + 50;
Write (B);

Read (A);
temp := A * 0.1;
A := A - temp;
Write (A);
Read (B);

B := B + temp;
Write (B);
```
Distributed Databases

- Distributed transaction management
  - The primary function of concurrency controller is to generate a serializable schedule for execution of a sequence of transactions — to devise algorithms that guarantee the generation of serializable schedules.
Distributed Databases

- Distributed transaction management

- In a distributed databases, with no data replication if local schedules are serializable then their union (global schedule) is also serializable as long as local serialization order is identical.
Distributed Databases

- Distributed transaction management
  - In distributed databases, in case of duplication, local schedules could be serializable, but the global schedule might not – mutual consistency of database is compromised.
  - Consider the following transactions and the two schedules:
Distributed Databases

- Distributed transaction management

\[ T_1: \]
- Read \((x)\)
- Store \(x \leftarrow x + 5\)
- Write \((x)\)
- Commit

\[ T_2: \]
- Read \((x)\)
- Store \(x \leftarrow x \times 10\)
- Write \((x)\)
- Commit

\[ S_1 = \{ R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2 \} \]

\[ S_2 = \{ R_2(x), W_2(x), C_2, R_1(x), W_1(x), C_1 \} \]
Distributed Databases

- Distributed transaction management
  - These local schedules are serializable. So each represent a correct execution order. However, at the global level they are run in reverse order.
  - Mutual consistency requires that all the values of all replicated data items be identical — one-copy serializable.
    - Each local schedule should be serializable,
    - Two conflicting operations should be in the same relative order in all of the local schedules when they appear together.
Distributed Databases

- **Distributed transaction management** — Concurrency control algorithms’ taxonomy

- Different parameters such as **degree of replication** (fully replication, no replication, partial replication) or **network topology** can be used to classify the concurrency control algorithms.
  - **Pessimistic algorithms** — Synchronizes The concurrent execution early.
  - **Optimistic algorithms** — delays synchronization until termination.
Distributed Databases

- Distributed transaction management — Concurrency control algorithms’ taxonomy

Concurrency Control Algorithms

- Pessimistic
  - Locking
    - Centralized
    - Primary Copy
    - Distributed
  - Timestamp Ordering
  - Hybrid

- Optimistic
  - Locking
    - Basic
    - Multi-version
  - Timestamp Ordering
    - Conservative
Distributed Databases

- Distributed transaction management — Lock-Based Protocol
  - One way to ensure serializability is to require data items to be accessed in a mutual exclusive fashion — while a transaction is accessing the data item, no other transaction can access that data item, i.e., being Locked.
Distributed Databases

- Distributed transaction management — Lock-Based Protocol

- There are various types of locks:
  - **Shared:** if a transaction $T_i$ has obtained a shared-mode lock (lock-s($Q$)) on item $Q$, then $T_i$ can read $Q$, but cannot write $Q$.
  - **Exclusive:** if a transaction $T_i$ has obtained an exclusive-mode lock (lock-x($Q$)) on item $Q$, then $T_i$ can read and write $Q$. 
Distributed Databases

- Distributed transaction management — Lock-Based Protocol
  - The following matrix shows the compatibility between different lock modes:

```
<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>X</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
```
Distributed Databases

- Distributed transaction management — Lock-Based Protocol

- To access a data item, transaction $T_i$ must first request for a lock on that data item. If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions are released.
Distributed Databases

- Distributed transaction management — Lock-Based Protocol
  - Assume the following two transactions:

```plaintext
Lock-X (B);
Read (B);
B := B – 50;
Write (B);
Unlock (B);
Lock-X (A);
Read (A);
A := A + 50;
Write (A);
Unlock (A);

Lock-S (A);
Read (A);
Unlock (A);
Lock-S (B);
Read (B);
Unlock (B);
Display (A+B);
```
Distributed Databases

Distributed transaction management

- Lock-Based Protocol

Incorrect Schedule, why?

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Lock</th>
<th>Grant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock (B)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Read (B)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>B := B - 50;</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Write (B)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Unlock (B)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lock (A)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Read (A)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A := A + 50;</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Write (A)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Unlock (A)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lock (B)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Read (B)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Unlock (B)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lock (A)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Read (A)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Unlock (A)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Display (A+B)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Grant (B, T1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Grant (A, T2)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Grant (B, T2)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Grant (A, T1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Grant (A, T2)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Grant (A, T1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Grant (B, T2)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Grant (A, T1)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Distributed Databases

- Distributed transaction management — Lock-Based Protocol

- Now assume the following similar transactions to the previous ones:

```
Lock-X (B);
Read (B);
B := B - 50;
Write (B);
Lock-X (A);
Read (A);
A := A + 50;
Write (A);
Unlock (B);
Unlock (A);
Lock-S (A);
Read (A);
Lock-S (B);
Read (B);
Display (A+B);
Unlock (A);
Unlock (B);
```

Scheduling these two will not result in a wrong sequence of Operations.
Distributed Databases

- Distributed transaction management — Lock-Based Protocol

- Unfortunately, locking can lead to deadlock, consider the partial schedule of previous transactions.

```
Lock-X (B);
Read (B);
B := B – 50;
Write (B);
Lock-S (A);
Read (A);
Lock-S (B);
Lock-X (A);
```
Distributed Databases

- Distributed transaction management – Lock-Based Protocol
  - We will define a set of rules, called locking protocol, to indicate when a transaction may lock and unlock a data item.
Distributed Databases

- **Distributed transaction management — Lock-Based Protocol**
  - Let \( \{ T_0, T_1, \ldots, T_n \} \) be a set of transactions in a schedule \( S \). \( T_i \) proceeds \( T_j \) in \( S \), written \( T_i \rightarrow T_j \), if there exist a common data item \( Q \) such that \( T_i \) has held a lock mode \( A \) on \( Q \), and \( T_j \) has held a lock mode \( B \) on \( Q \) later, and \( \text{comp} (A, B) = \text{false} \). Then in any equivalent serial schedule \( T_i \) must appear before \( T_j \).
  - In another words, the **precedence rule**, implies data dependence between the two transactions. Conflicts between instructions implies incompatibility of lock modes.
Distributed Databases

- **Distributed transaction management** — Lock-Based Protocol
  
  - Within the scope of locking protocol, one has to be concerned about **starvation**. Starvation can be avoided by the concurrency control manager.
  
  - Assume $T_i$ request a lock on a data item $Q$ in a particular mode $M$, the lock is granted if:
    
    - There is no other transaction holding a lock on $Q$ in a mode that conflicts with $M$.
    
    - There is no other transaction waiting for a lock on $Q$ and made its lock request before $T_i$.

  - In short, a lock request will never get blocked by a lock request that is made later.
Distributed Databases

- Last lecture
  - Conflict equivalence and conflict serializability
  - anomalies
  - Different classes of Concurrency control algorithms
  - Lock-based protocol
  - Shortcomings of lock-based protocol
    - Deadlock
    - Starvation
    - Incorrect Schedule
    - Cascading aborts
Exam #2

- Give two reasons in support of “equivalence rule 1” i.e., cascade of selections can be deconstructed into a sequence of individual selections; $\sigma_{\theta_1 \land \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$.

  - Allows to push selection operator down the query tree (see page 69 or 89).

  - If the selection is on the attribute that allows access to the tuples (see page 109).
Consider the following two relations:

- S (S_id: integer, S_name: string, rating: integer, age: real)
- R (S_id: integer, bid: integer, day: dates, r_name: string)

Further assume the following query:

```sql
SELECT S.S_name
FROM R, S
WHERE R.S_id = S.S_id
    AND R.bid = 100
    AND S.rating > 5
    AND R.day = '8/9/09'
```

What is a query plan for the aforementioned SQL query assuming that the underlying platform allows pipeline execution of operations (justify your choice of the plan)?
\[
\begin{align*}
\Pi_{\text{Sname}} & \quad \text{On the fly} \\
\sigma_{\text{rating} > 5} & \quad \text{On the fly} \\
\sigma_{\text{bid} = 100} & \quad \text{On the fly} \\
\sigma_{\text{day} = 8/9/09} & \\
S & \\
R & 
\end{align*}
\]
Why left deep join strategy?

- Left-deep trees allow us to generate all fully pipelined plans; that is, plans in which all join operations are evaluated using pipelining.

- As another advantage, having the base relation as one of the inputs to join allows the optimizer to utilize any access paths on that relation that may be useful in executing the join.
Distributed Databases

An example:

\[ E_1 \bigcap_\theta (E_2 - E_3) = (E_1 \bigcap_\theta E_2 - E_1 \bigcap_\theta E_3). \]

Let us rename \((E_1 \bigcap_\theta (E_2 - E_3))\) as \(R_1\), \((E_1 \bigcap_\theta E_2)\) as \(R_2\) and \((E_1 \bigcap_\theta E_3)\) as \(R_3\).

It is clear that if a tuple \(t\) belongs to \(R_1\), it will also belong to \(R_2\).

If a tuple \(t\) belongs to \(R_3\), \(t[E_3’s\ attributes]\) will belong to \(E_3\), hence \(t\) cannot belong to \(R_1\).

From these two we can say that \(\forall t, \ t \in R_1 \Rightarrow t \in (R_2 - R_3)\)

It is clear that if a tuple \(t\) belongs to \(R_2 - R_3\), then \(t[R_2’s\ attributes] \in E_2\) and \(t[R_2’s\ attributes] \not\in E_3\).

Therefore: \(\forall t, \ t \in (R_2 - R_3) \Rightarrow t \in R_1\)

The above two equations imply the given equivalence.
Distributed Databases

- Distributed transaction management — Two-phase Locking Protocol

  - This protocol ensures serializability, however, it requires that each transaction issue lock and unlock requests in two phases:
    - **Growing Phase**: A transaction may obtain locks, but may not release any lock.
    - **Shrinking phase**: A transaction may release locks, but may not obtain any new locks.

  - Initially, a transaction is in the growing phase. It acquires locks as needed. Once it releases a lock, it enters the shrinking phase, and can issue no more lock requests.
Distributed Databases

- Distributed transaction management — Two-phase Locking Protocol

![Diagram showing the growing and shrinking phases of a transaction](image-url)
Distributed Databases

- **Distributed transaction management** — Two-phase Locking Protocol

- The following two transactions are not 2-phase:

  ```
  Lock-X (B);
  Read (B);
  B := B – 50;
  Write (B);
  Unlock (B);
  Lock-X (A);
  Read (A);
  A := A + 50;
  Write (A);
  Unlock (A);
  ```

  ```
  Lock-S (A);
  Read (A);
  Unlock (A);
  Lock-S (B);
  Read (B);
  Unlock (B);
  Display (A+B);
  ```
Distributed Databases

- **Distributed transaction management — Two-phase Locking Protocol**

- The following two transactions are 2-phase:

```
Lock-X (B); Read (B); B := B - 50; Write (B);
Lock-X (A); Read (A); A := A + 50; Write (A);
Unlock (B); Unlock (A);
Lock-S (A); Read (A); Lock-S (B); Read (B);
Display (A+B); Unlock (A); Unlock (B);
```
Distributed Databases

- Distributed transaction management — Two-phase Locking Protocol
  - For a 2-phase transaction, the point where the transaction obtains its last lock is called the lock point of the transaction.
  - In a two phase locking protocol, transactions can be scheduled (ordered) based on their lock points (homework #3).
Distributed Databases

- Distributed transaction management — Two-phase Locking Protocol
  - Two phase locking protocol does not ensure freedom from deadlock.

These two transactions are 2-phase, but in this schedule they are deadlocked.

```
Lock-X (B);
Read (B);
B := B – 50;
Write (B);

Lock-S (A);
Read (A);
Lock-S (B);
Lock-X (A);
```
Distributed Databases

- Distributed transaction management — Two-phase Locking Protocol
  - A “good” schedule should also be cascadeless. Cascading rollback may occur under two phase locking protocol.
  - Look at the following schedule and the reason why rollback cascading must be enforced.
Distributed Databases

- **Distributed transaction management** — Two-phase Locking Protocol

If the first transaction fails after this point, then the other two transactions have to be rolled back.
Distributed Databases

- **Summary**
  - Testing for Serializability
  - Concurrency control
  - Lock based protocol
  - Two-phase locking protocol
  - Deadlock
  - Cascading rollback
Distributed Databases

- Distributed transaction management — Two-phase Locking Protocol
  - Two-phase locking protocol can be modified to avoid cascading rollback:
    - Strict two-phase locking protocol,
    - Rigorous two-phase locking protocol.
Distributed Databases

- Distributed transaction management — Strict two-phase locking protocol
  - Within a two-phase locking protocol, this protocol requires all exclusive-locks be held until transaction commits.
  - Any data written by an uncommitted transaction are locked and unaccessible to any other transactions to read it.
Distributed Databases

- Distributed transaction management — Strict two-phase locking protocol

![Diagram showing growing and shrinking phases of transaction duration with number of locks and data items used](image-url)
Distributed Databases

- Distributed transaction management — Rigorous two-phase locking protocol
  - Within a two-phase locking protocol, this protocol requires all locks be held until transaction commits.
  - Transactions can be serialized in the order in which they commit.
Distributed Databases

- **Distributed transaction management — Lock conversions**
  - The basic two-phase locking can be extended to allow a better performance.
  - Consider the following two transactions:

```
Read (a_1); Read (a_2);
•
•
•
Read (a_n);
Write (a_1);
```
```
Read (a_1);
Read (a_2);
Display (a_1+a_2);
```
Distributed Databases

- **Distributed transaction management — Lock conversions**

- In a normal two-phase locking protocol, first transaction locks $a_I$ in exclusive mode, as a result the second transaction must be scheduled after execution of the first one (serial schedule).

- However, first transaction needs exclusive lock on $a_I$ towards the end of its operations.

- If we allow $a_I$ to be locked in the shared mode initially, then the second transaction can be scheduled concurrent with the first one.
Distributed Databases

- **Distributed transaction management — Lock conversions**
  - The two-phase locking protocol can be extended by allowing the lock conversion.
  - We will allow to *upgrade* a lock to exclusive mode and *downgrade* a lock from the exclusive mode.
  - We also impose the following restriction, upgrading can be done during the growing phase and downgrading can be done during the shrinking phase.
Distributed Databases

- **Distributed transaction management — Lock conversions**

```
Lock-S (a_1);
Lock-S (a_2);
Lock-S (a_3);
Lock-S (a_4);
•
•
•
Lock-S (a_n);
Upgrade (a_1);
```

- Note a transaction attempting to upgrade a lock on a data item may be forced to wait if the data item is currently locked by another transaction in shared mode.
Distributed Databases

- Distributed transaction management — Lock conversions
  - A two-phase locking protocol enhanced by lock conversion generates only conflict serializable schedules (transactions can be serialized based on their lock points).
Distributed Databases

- **Distributed transaction management** — Two-phase Locking Protocol
  
  - For a set of transactions, there may be conflict serializable schedules that cannot be obtained through the two-phase locking protocol.
Distributed Databases

- Distributed transaction management — Two-phase Locking Protocol

- A simple automated scheme can be used to generate lock and unlock instructions for an arbitrary transaction:
  - When a transaction issues a read (Q), the system issues a lock-s (Q) instruction followed by the read (Q) instruction.
  - When a transaction issues a write (Q), the system check to see whether the same transaction holds a shared lock on Q. If it does, then an upgrade (Q) instruction followed by the write (Q) instruction is issued. Otherwise, a lock-x (Q) followed by write (Q) is issued.
  - All locks obtained by the transaction are unlocked after the transaction commit or aborts.
Distributed Databases

- Distributed transaction management — Graph-based Protocol
  - Knowledge about the access order to the database items can be used to schedule transactions.
  - Such a knowledge can be used to develop a locking protocol that is not two-phase.
Distributed Databases

- **Distributed transaction management — Graph-based Protocol**
  - We impose a partial ordering $\rightarrow$ on the set of the data items $D = \{d_1, d_2, \ldots, d_n\}$.
  - If $d_i \rightarrow d_j$, then any transaction accessing both $d_i$ and $d_j$ must access $d_i$ before $d_j$.
  - This partial ordering implies that the set $D$ can be viewed as a directed acyclic graph — database graph.
Distributed Databases

- **Distributed transaction management — Tree-based Protocol**

  - In the tree protocol, only lock-x operation is allowed. Each transaction can lock a data item based on the following rules:
    - The first lock may be on any data item.
    - Subsequently, a data item can be locked only if its parent is currently locked by the same transaction.
    - Data items may be unlocked at any time.
    - A data item that has been locked and unlocked cannot be relocked again.
Distributed Databases

- **Distributed transaction management** — Tree-based Protocol

  - It can be shown that all legal transactions under the tree-based protocol are conflict serializable.
  - Consider the following database graph and four transactions:
Distributed Databases

- Distributed transaction management – Tree-based Protocol

Database Graph
Distributed Databases

- Distributed transaction management –
  Tree-based Protocol

- Lock-x (B);
- Lock-x (E);
- Lock-x (D);
- Unlock (B);
- Unlock (E);
- Lock-x (G);
- Unlock (D);
- Unlock (G);
- Lock-x (D);
- Lock-x (H);
- Unlock (D);
- Unlock (H);
- Lock-x (B);
- Lock-x (E);
- Lock-x (H);
- Unlock (E);
- Unlock (D);
- Unlock (H);
- Lock-x (D);
- Lock-x (H);
- Unlock (D);
- Unlock (H)
Distributed Databases

- Distributed transaction management – Tree-based Protocol

Diagram:

```
  A -> B -> D -> H
  C -> E -> F -> G -> I
  J
```

Transactions:

- Lock-x (B); Lock-x (D); Unlock (D);
- Lock-x (E); Unlock (B); Unlock (E);
- Lock-x (G); Unlock (D);
- Unlock (G);
- Unlock (H);
- Lock-x (B); Lock-x (E);
- Lock-x (D); Unlock (D);
- Lock-x (H); Unlock (D);
- Unlock (E); Unlock (H);
- Unlock (B);
- Lock-x (D);
- Lock-x (H);
- Unlock (D);
- Unlock (H);
- Unlock (B);
Distributed Databases

- **Distributed transaction management — Tree-based Protocol**
  - Tree-based protocol ensures conflict serializability and freedom from deadlock.
  - Tree-based protocol does not ensure recoverability and cascadelessness.
  - Recoverability and cascadelessness can be ensured at the expense of performance if locks are not released until the end of transaction.
Distributed Databases

- Last lecture
  - Two-phase locking protocol and its variations
  - How to improve the basic Two-phase locking protocol
Distributed Databases

- Distributed transaction management — Timestamp-based Protocol
  - To each transaction $T_i$ a unique fixed timestamp $TS(T_i)$ is associated. Timestamp could be:
    - The system clock,
    - A logical counter that is incremented each time a timestamp is associated to a transaction.
  - The timestamps of transactions determine the serializability order — if $TS(T_i) < TS(T_j)$ then the system must ensure that the generated schedule is equivalent to a serial schedule in which transaction $T_i$ appears before $T_j$. 
Distributed Databases

- Distributed transaction management – Timestamp-based Protocol

- To implement timestamp-based protocol, two timestamp values are associated with each data item:
  - \( W\text{-timestamp}(Q) \) denotes the largest timestamp of any transaction that executed Write\((Q)\) successfully.
  - \( R\text{-timestamp}(Q) \) denotes the largest timestamp of any transaction that executed Read\((Q)\) successfully.

- These values are updated whenever a new Read\((Q)\) or Write\((Q)\) is executed.
Distributed Databases

- **Distributed transaction management – Timestamp-based Protocol**
  - In case $T_i$ issues a read($Q$):
    - If $TS(T_i) < W$-timestamp($Q$), $T_i$ should have read the old value of $Q$ that has been modified. Hence, read operation is rejected and $T_i$ is rolled back.
    - If $TS(T_i) \geq W$-timestamp($Q$), the read operation is executed and $R$-timestamp($Q$) is set to the maximum of $R$-timestamp($Q$) and $TS(T_i)$. 
Distributed Databases

- Distributed transaction management — Timestamp-based Protocol
- In case $T_i$ issues a write($Q$):
  - If $TS(T_i) < R$-timestamp($Q$), the value of $Q$ generated by $T_i$ is relatively old, write is rejected and $T_i$ is rolled back.
  - If $TS(T_i) < W$-timestamp($Q$), then $T_i$ is trying to write an outdated value to $Q$, write is rejected and $T_i$ is rolled back.
  - Otherwise, write is executed and $W$-timestamp($Q$) is set to $TS(T_i)$. 
Distributed Databases

- **Distributed transaction management – Timestamp-based Protocol**

- **Consider the following transactions:**
  
  **Green Transaction:**
  
  `Read (B);`  
  `Read (A);`  
  `Display (A+B);`  

  **Red Transaction:**
  
  `Read (B);`  
  `B := B - 50;`  
  `Write (B);`  
  `Read (A);`  
  `A := A + 50;`  
  `Write (A);`  
  `Display (A+B);`

- **Note the timestamp of the green transaction is less than the timestamp of the red transaction.**
Distributed Databases

- Distributed transaction management –
  Timestamp-based Protocol

Read (B);
Read (A);
Display (A+B);

Read (B);
B := B – 50;
Write (B);
Read (A);
A := A + 50;
Write (A);
Display (A+B);
Distributed Databases

- **Distributed transaction management** – Timestamp-based Protocol
  - Note there are schedules that are possible under two-phase locking that are not possible under timestamp protocol and vice versa.
  - **Timestamp protocol** ensures conflict serializability and freedom from deadlock, but it may cause starvation of long transactions by conflicting short transactions.
Distributed Databases

- Distributed transaction management — Timestamp-based Protocol
  - Timestamp-ordering does not cause deadlock, since transactions never wait while they have access rights to data items.
  - The penalty of deadlock free comes at the expense of potential restart of a transaction again and again.
  - Operations from the scheduler is sent to the database processor one at a time. As long as an operation is not terminated a new one will not be passed on to the processor.
Distributed Databases

- **Distributed transaction management**
  - Within the scope of the distributed databases, we distinguish two entities:
    - Local entities, and
    - Global entity.
  - As a result, we can talk about local transactions vs. global transactions.
Distributed Databases

- **Distributed transaction management**
  - In this environment, each site has its own local transaction manager, whose function is to ensure ACID properties for those local transactions executed at that site.
  - Different transaction managers cooperate with each other to execute global transactions.
Distributed Databases

- Centralized Transaction Execution

User application

Transaction Manager

Scheduler

Recovery Manager

Scheduled operations

Operations

Result
Distributed Databases

- **Centralized Transaction Execution**
  - **Transaction Manager** is responsible for coordinating the execution of the database operations on behalf of an application.
  - **Scheduler** is responsible for the implementation of a specific concurrency control algorithm.
  - **Recovery manager** is responsible for implementing procedures that transform the database into a consistent state after a failure.
Distributed Databases

- Distributed transaction management

General configuration

Site_1

Spring 13
Distributed Databases

- **Distributed transaction management** — General configuration
  - **Transaction manager** manages the execution of transactions (local transaction or global sub-transaction) that access local data.
  - **Transaction coordinator** coordinates the execution of transactions (local or global) that are initiated at that site.
Distributed Databases

- **Distributed transaction management** – General configuration

- The structure of transaction manager is similar to the one in a centralized database. It is responsible to:
  - Maintain a log for recovery,
  - Participate in a concurrency control protocol to coordinate concurrent execution of transaction at that site.
Distributed Databases

- Distributed transaction management — General configuration
  - The transaction coordinator is a new entity and responsible for:
    - Starting the execution of the transaction,
    - Converting a global transaction into sub-transactions and distribute sub-transactions to the designated sites, and
    - Coordinate the termination of the transactions — note a global transaction must be committed or aborted at all sites.
Distributed Databases

- Distributed transaction management — General configuration
  - Besides the type of failures common in a centralized database (software/hardware errors, disk crashes), a distributed system suffers from:
    - Site failure,
    - Loss/corruption of messages,
    - Link failure,
    - Network partitioning
Distributed Databases

- **Distributed transaction management** — General configuration

- Within a distributed system, sites are communicating with each other via messages. If two sites are not physically linked together, then messages must be routed through a sequence of communication links.

- In case of link failure, messages might be able to find different route through the network.
Distributed Databases

- **Distributed transaction management** — General configuration
  
  - Network partitioning is the result of link failure, where a group of sites cannot communicate with each other — system is partitioned into subsystems.
Distributed Databases

- **Distributed transaction management** — General Comments
  - **Lock Based approach**: Transactions are synchronized by physical or logical locks on some portion (granule) of the database — Locking granularity.
  - In a distributed environment, we can distinguish three classes of lock based protocols:
    - Centralized Locking
    - Primary Copy Locking
    - Decentralized Locking
Distributed Databases

- Distributed transaction management — Lock Based approach
  - Centralized Locking: One of the sites in the network is designated as the primary site, where the lock tables for the entire database are stored, in charge with the task of granting locks to transactions.
Distributed Databases

- Distributed transaction management — Lock Based approach
  - Primary Copy Locking: One of the copies of each data unit is designated as the primary copy and it is the primary copy that has to be locked for the purpose of accessing that data unit — all transactions desiring to access data item obtain their lock at the site where the primary copy resides at.
  - If data item is not replicated, the primary copy protocol distributes the lock management task among a number of sites.
Distributed Databases

- **Distributed transaction management** — Lock Based approach
  - **Decentralized Locking**: The lock manager duty is shared by all the sites in the network — Execution of a transaction involves the participation and coordination of schedulers at several sites.
  - In case of replication, transaction accessing a data item must obtain locks at all sites.
Distributed Databases

- Distributed transaction management — General Comments
  - Timestamp Ordering: Assigns a unique identifier to each transaction and data items in order to organize their execution sequence.
  - In this group we can talk about:
    - Basic Timestamp Ordering
    - Multi-version Timestamp Ordering
    - Conservative Timestamp Ordering
Distributed Databases

- Distributed transaction management — General Comments

- Note there is a class of Hybrid Concurrency control algorithms which are mixed of locking-based and timestamp-based schemes. This class is intended to improve efficiency and the level of concurrency.
Distributed Databases

- **Distributed transaction management** — Centralized Two-phase Locking protocol
  - Lock manager responsibility is delegated to a single site.
  - Transaction coordinators at the other sites communicate with the centralized lock manager rather than with their own lock managers.
  - This scheme also is referred to as: primary site two-phase locking.
Distributed Databases

- Distributed transaction management — Centralized Two-phase Locking protocol

Data Processing at Participating sites

Coordinating Transaction Manager

Central site Lock Manager

1 Lock request

3 Operations

2 Lock granted

4 End of Operations

5 Release Locks
Distributed Databases

- **Distributed transaction management** — Centralized Two-phase Locking protocol
  - Centralized lock manager does not communicate with the data processing sites directly.
  - The distributed transaction manager must implement replica control protocol, if database is replicated.
  - Bottleneck at central lock manager and reliability are the major drawbacks of this approach.
Distributed Databases

- **Distributed transaction management** — Primary copy

  **Two-phase Locking protocol**

  - This scheme is the direct extension of centralized two-phase locking protocol in an attempt to remove its performance bottleneck.
  
  - It simply distributes the task of lock manager among several lock managers for a given set of lock units.
  
  - The transaction coordinators send their lock and unlock requests to the lock managers that are responsible for specific lock unit — The location of the primary copy of each data item needs to be determined before sending a lock or unlock request.
Distributed Databases

- Distributed transaction management — Distributed Two-phase Locking protocol
  - Here a lock manager exists at each site.
    - In case of no replication, this scheme degenerates into the Primary copy Two-phase Locking protocol.
    - In case of replica, the algorithm implements the read once/write all (ROWA) replica control protocol.
Distributed Databases

- Distributed transaction management — Distributed Two-phase Locking protocol

- Coordinating Transaction Manager
- Participating schedulers Lock Managers
- Participating Data Processing sites

1 Operations (Lock requests)
2 Operations
3 End of Operations
4 Release Locks
Distributed Databases

- **Distributed transaction management** — Distributed Two-phase Locking protocol
- In comparison to centralized approach, lock and unlock messages are sent to the lock managers at all participating sites. In addition, operations are sent, by participating lock managers, to the data processors instead of coordinating transaction manages.
Distributed Databases

- **Distributed transaction management — Two-phase commit protocol**

  - Transaction $T$ is initiated at site $S_i$ with the transaction coordinator $TC_i$.
  - $T$ is decomposed by $TC_i$ and transmitted to different sites for execution.
  - When all sites at which $T$ is executed inform $TC_i$ that $T$ is completed, $TC_i$ starts the 2-phase commit protocol.
Distributed Databases

- **Distributed transaction management — Two-phase commit protocol (Phase 1)**
  - $TC_i$ adds the record “prepare $T$” to the log and forces the log to permanent storage.
  - “Prepare $T$” message is communicated with all the sites involved.
    - At each designated site, the transaction manager follows the following actions:
      - If not willing to “commit”, then adds “no $T$” to its log and sends an “abort $T$” message to $TC_i$.
      - If willing to commit, then adds “ready $T$” to its log and sends a “ready $T$” message to $TC_i$. 
Distributed Databases

- **Distributed transaction management — Two-phase commit protocol (Phase2)**
  - On receiving replies from the sites, $TC_i$ determines whether or not the transaction must be committed or aborted.
  - Based on the local decisions, either a “commit $T$” or an “abort $T$” is logged on permanent storage.
  - $TC_i$ sends either an “abort $T$” or “commit $T$” to the participating sites.
  - The message from the $TC_i$ is logged at the participating sites.
Distributed Databases

- **Distributed transaction management — Two-phase commit protocol**
  - A “ready T” message from a participating site to $TC_i$ is a promise that the site will follow the coordinator commit or abort command.
  - **Unanimity** is required by the coordinator to commit a transaction.
  - In some implementation, at the end of the 2$^{nd}$ phase, participating sites send “acknowledge T” message to the $TC_i$. $TC_i$ upon receiving all “acknowledge T” messages, adds the “complete T” record into its log.
Distributed Databases

- Last lecture
  - Timestamp based protocol and its variations
  - Transaction processing in distributed systems
  - Lock-based protocol in distributed system
    - Centralized configuration
    - Primary copy configuration
    - Decentralized configuration
  - Two-phase commit protocol
Distributed Databases

- Distributed transaction management — Two-phase commit protocol
  - Failure of a participating site: Transaction is aborted if $TC_i$ detects a failure before receiving “ready T” message, otherwise the coordinator continue normal sequence of operations.
Distributed Databases

- **Distributed transaction management — Two-phase commit protocol**

  - After recovery from failure, the site must examine its log to determine the fate of those transactions that were in the midst of execution when the failure occurred.
    - If log contain “commit $T$”, then the site executes “redo ($T$)”,
    - If log contains “abort $T$”, then the site performs “undo ($T$)”,
    - If log contains “ready $T$”, then it should consult with the coordinator,
    - If log contains no “commit, abort, ready” messages about $T$, then it performs “undo ($T$)”.
Distributed Databases

- Distributed transaction management — Two-phase commit protocol
  - Failure of the coordinator: If the coordinator fails in the midst of operation of a transaction $T$:
    - If an active site contains “commit T” in its log, then $T$ must be committed,
    - If an active site contains “abort T” in its log, then $T$ must be aborted,
    - If some active site do not contain “ready T” in their logs, then $T$ must be aborted,
    - If none of these cases holds, then all active sites must have “ready T” record in their logs. In this case they must wait for the coordinator site to recover — this will cause blocking problem since data items involved in $T$ might be unavailable to other transactions.
Distributed Databases

- Distributed transaction management — Two-phase commit protocol
  - Network Partition: In this case we have two possibilities:
    - Coordinator and all sites are in the same partition. In this case the failure has no effect on the fate of transaction.
    - Coordinator and all sites involved belong to several partitions. In this case, coordinator and sites in the same partition, follow the protocol that the other sites in other partitions have failed. Sites in the partitions that does not contain the coordinator follow the protocol that the coordinator has failed.
Distributed Databases

- Distributed transaction management — Three-phase commit protocol
  - This is an extension to two-phase commit that avoids blocking under certain assumptions. It assumes no network partitioning and can tolerate up to $K$ sites failure.
  - Under aforementioned assumptions, it introduces a third phase where multiple sites are involved in the decision to commit.
Distributed Databases

- **Distributed transaction management** — Three-phase commit protocol
  - Instead of noting the commit decision in its log (permanent storage) and then informing other sites involved, the coordinator make sure that at least $k$ sites are aware of its intension to commit.
  - If the coordinator fails, the remaining sites first select a coordinator, the new coordinator, checks the status of the transaction from other sites. If there was a decision to commit the transaction, the new coordinator respects that and initiate the third phase, otherwise, it will abort the transaction.
Distributed Databases

- Distributed transaction management – Locking protocols
  - First we will look at the schemes that require update to be done on all replicated data. Then, we will look at schemes that allow us to relax this restriction.
Distributed Databases

- Distributed transaction management –
  Locking protocol
  - Locking protocol as we studied before can be used for distributed environment by extending the scope of lock-manager in order to handle replicated data.
  - Two cases will be considered:
    - Single lock-manager approach
    - Distributed lock-manager approach
Distributed Databases

- **Distributed transaction management** — Single lock-manager approach
  - This is the same as locking scheme in centralized database environment.
  - A *single lock-manager* is maintained in one site — say $S_i$. As a result, every lock and unlock requests are made at site $S_i$. 
Distributed Databases

- **Distributed transaction management** — Single lock-manager approach

  - Upon a request, lock-manager determines whether or not the lock can be granted immediately:
    - If so, a message to that effect is sent to the site requesting the lock.
    - If no, request is delayed until it can be granted and a message to this fact is sent to the site requesting the lock.

  - A transaction can read data from any replica, but all replicas participate in a write operation.
Distributed Databases

- Distributed transaction management — Single lock-manager approach
  - Simple implementation
  - Simple deadlock detection
    - Are major advantages of this approach.
  - Bottleneck at the lock-manager
  - Vulnerability and lack of fault tolerance
    - Are the disadvantages of this approach.
Distributed Databases

- **Distributed transaction management** — Distributed lock-manager approach
  - Each site maintains a **local lock manager** whose function is to administer the lock and unlock requests for data items stored at that site.
Distributed Databases

- **Distributed transaction management** — Distributed lock-manager approach (request for unreplicated data)
  - When a transaction wishes to lock data item from a site ($S_i$) and if the data item is not replicated, a message is sent to the lock manager at site $S_i$.
    - If the data item is locked in an incompatible mode, then the request is delayed until it can be granted.
    - Once it is determined that the lock can be granted, the lock manager sends a message back to the initiator that the lock request is granted.
Distributed Databases

- **Distributed transaction management** — Distributed lock-manager approach (request for unreplicated data)
  - This approach has low implementation overhead, it is easy to implement, and without bottleneck at a local site.
  - However, it is harder to implement deadlock detection — there is a potential for **inter-site deadlock** even when there is no deadlock within a single site.
Distributed Databases

- Distributed transaction management — Distributed lock-manager approach (request for replicated data)
- Primary copy protocol
  - In this case, one replica is chosen as the primary copy and hence, its corresponding site is called primary site.
  - Any lock request for a data item must be sent to the primary site. As a result, this approach allows implementation of concurrency control for replicated data as the replica does not exist.
Distributed Databases

- **Distributed transaction management** — Distributed lock-manager approach (request for replicated data)
- **Majority protocol**
  - In case data item is replicated in \( n \) sites, a lock request must be sent to more than one-half of the sites.
  - A transaction cannot operate on the requested data item until it has obtained a lock on the majority of the replicas.
Distributed Databases

- **Distributed transaction management** — Distributed lock-manager approach (request for replicated data)
- **Majority protocol**
  - This approach is more complicated to implement and requires more messages to lock and unlock a data item.
  - There is also a potential for global deadlocks.
Distributed Databases

- Distributed transaction management — Distributed lock-manager approach (request for replicated data)
- Biased protocol
  - This is a version of the majority protocol where the request for shared locks are given more favorable treatment than the request for exclusive locks.
  - Consequently, it imposes less overhead on read operations than does the majority protocol — it is more appropriate for application domains which require much more read operations than write operations.
  - As before, the transaction does not operate on a data item until it has successfully obtained a lock on a majority of the replica.
Distributed Databases

- **Distributed transaction management** — Distributed lock-manager approach (request for replicated data)
  - **Biased protocol**
    - **Shared lock** request on a data item is sent to just one site holding a replica.
    - **Exclusive lock** request on a data item is sent to all sites holding the replicas.
Distributed Databases

- Distributed transaction management — Distributed lock-manager approach (request for replicated data)
- Quorum Consensus protocol
  - It is a generalization of both majority and biased protocols.
  - Each site is assigned a weight, $w_i$.
  - Read and write operations on an item $x$ is enhanced by two integers, read Quorum $Q_r$ and write Quorum $Q_w$, that must satisfy the following relations:
    
    $$Q_r^x + Q_w^x > \sum w_i^x$$
    $$2*Q_w^x > \sum w_i^x$$
Distributed Databases

- Distributed transaction management — Distributed lock-manager approach (request for replicated data)
- Quorum Consensus protocol
  - To execute a read/write on $x$, enough replicas must be read/written that their total weight satisfy the following:

$$\sum w_i^x \geq Q_r^x / Q_w^x$$
Distributed Databases

- Distributed transaction management — Distributed lock-manager approach (request for replicated data)

- Quorum Consensus protocol
  - This approach is more dynamic which allows one, based on the application domain, to favor read or write operations.
  - In addition, site weight can be assigned such that the sites that are more reliable weighted higher.
Distributed Databases

- Distributed transaction management — Conservative timestamp-ordering
  - What is the major problem with the timestamp-ordering in a distributed system?
  - Timestamp-ordering is a deadlock free protocol, since operations never wait, but it forces transaction restart.
  - This is a major problem in distributed systems since transactions generated by inactive sites executed at active sites will keep being rejected continuously.
Distributed Databases

- **Distributed transaction management — timestamp-ordering**
  - To avoid continuous restart of rejected transactions, counters at different sites must be synchronized.
  - Synchronization cost of counters is expensive — large number of required messages.
  - Simple solution can be adapted to avoid high cost of synchronization (if we use system clock and if clocks are of comparable speed).
Distributed Databases

- **Distributed transaction management — timestamp-ordering**
  - As another solution, we let the transaction coordinators to communicate with each other.
  - A transaction coordinator sends its remote operations to other transaction coordinators at other sites — instead of transaction managers.
  - At receiving sites, each transaction coordinator whose counter is less than the incoming timestamp, adjust its counter to one more than the incoming one — this policy ensures that none of the counters gets away or lags behind the others significantly.
Distributed Databases

- Distributed transaction management — Timestamp ordering
  - This approach assigns a unique identifier to each transaction in order to decide the serialization order. So in a distributed environment the challenge lies in the generation of the unique identifier.
  - Two cases can be recognized:
    - Centralized timestamping,
    - Distributed timestamping.
Distributed Databases

- **Distributed transaction management** — Timestamp ordering
  - **Centralized timestamping**: In this case, a single site distributes the timestamping.
  - **Distributed timestamping**: In this case, a global timestamp is composed of two entities:
    - A unique local timestamp, as in centralized environment,
    - A unique site identifier.

| Local site timestamp | Site identifier |