Outline

- Introduction & architectural issues
- Data distribution
- Distributed query processing
- Distributed query optimization
  - Distributed transactions & concurrency control
    - Transaction models and concepts
    - Distributed concurrency control
  - Distributed reliability
- Data replication
- Parallel database systems
- Database integration & querying
- Peer-to-Peer data management
- Stream data management
- MapReduce-based distributed data management

Transaction

A transaction is a collection of actions that make consistent transformations of system states while preserving system consistency.

- concurrency transparency
- failure transparency

![Diagram of transaction lifecycle]

- Database in a consistent state
- Database may be temporarily in an inconsistent state during execution
- Database in a consistent state
**Transaction Example – A Simple SQL Query**

Transaction BUDGET_UPDATE

begin

EXEC SQL
UPDATE PROJ
SET BUDGET = BUDGET * 1.1
WHERE PNAME = "CAD/CAM"

decl.

**Example Database**

Consider an airline reservation example with the relations:

FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
CUST(CNAME, ADDR, BAL)
FC(FNO, DATE, CNAME, SPECIAL)
Example Transaction – SQL Version

**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. {Reservation}

Termination of Transactions

**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
    **output**("no free seats");
    **Abort**
  else
    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")
  endif
end. {Reservation}
Example Transaction – Reads & Writes

```plaintext
Begin_transaction Reservation
begin
  input(flight_no, date, customer_name);
  temp ← Read(flight_no(date).stsold);
  if temp = flight(date).cap then
    begin
      output("no free seats");
      Abort
    end
  else begin
    Write(flight(date).stsold, temp + 1);
    Write(flight(date).cname, customer_name);
    Write(flight(date).special, null);
    Commit;
    output("reservation completed")
  end
end {Reservation}
```

Characterization

- **Read set (RS)**
  - The set of data items that are read by a transaction
- **Write set (WS)**
  - The set of data items whose values are changed by this transaction
- **Base set (BS)**
  - RS ∪ WS
Principles of Transactions

**Atomicity**
- all or nothing

**Consistency**
- no violation of integrity constraints

**Isolation**
- concurrent changes invisible $\Rightarrow$ serializable

**Durability**
- committed updates persist

---

**Workflows**

- “A collection of tasks organized to accomplish some business process.”
- **Types**
  - Human-oriented workflows
    - Involve humans in performing the tasks.
    - System support for collaboration and coordination; but no system-wide consistency definition
  - System-oriented workflows
    - Computation-intensive & specialized tasks that can be executed by a computer
    - System support for concurrency control and recovery, automatic task execution, notification, etc.
  - Transactional workflows
    - In between the previous two; may involve humans, require access to heterogeneous, autonomous and/or distributed systems, and support selective use of ACID properties
Workflow Example

$T_1$: Customer request obtained
$T_2$: Airline reservation performed
$T_3$: Hotel reservation performed
$T_4$: Auto reservation performed
$T_5$: Bill generated

Transactions Provide...

- **Atomic** and **reliable** execution in the presence of failures
- **Correct** execution in the presence of multiple user accesses
- Correct management of **replicas** (if they support it)
Transaction Processing Issues

- Transaction structure (usually called transaction model)
  - Flat (simple), nested
- Internal database consistency
  - Semantic data control (integrity enforcement) algorithms
- Reliability protocols
  - Atomicity & Durability
  - Local recovery protocols
  - Global commit protocols

Transaction Processing Issues

- Concurrency control algorithms
  - How to synchronize concurrent transaction executions (correctness criterion)
  - Intra-transaction consistency, isolation
- Reliability protocols
  - Atomicity & Durability
  - Local recovery protocols
  - Global commit protocols
- Replica control protocols
  - How to control the mutual consistency of replicated data
  - One copy equivalence and ROWA
Architecture Revisited

Centralized Transaction Execution
Distributed Transaction Execution

Concurrency Control

- The problem of synchronizing concurrent transactions such that the consistency of the database is maintained while, at the same time, maximum degree of concurrency is achieved.

- Anomalies:
  - Lost updates
    - The effects of some transactions are not reflected on the database.
  - Inconsistent retrievals
    - A transaction, if it reads the same data item more than once, should always read the same value.
Isolation Example

Consider the following two transactions:

\[ T_1: \text{Read}(x), \ x \leftarrow x+1, \ \text{Write}(x), \ \text{Commit} \]
\[ T_2: \text{Read}(x), \ x \leftarrow x+1, \ \text{Write}(x), \ \text{Commit} \]

Possible execution sequences:

\[ T_1: \text{Read}(x), \ T_2: \text{Read}(x) \]
\[ T_1: \ x \leftarrow x+1, \ T_2: \ x \leftarrow x+1 \]
\[ T_1: \text{Write}(x), \ T_2: \text{Write}(x) \]
\[ T_1: \text{Commit}, \ T_2: \text{Commit} \]

Execution History (or Schedule)

An order in which the operations of a set of transactions are executed.

A history (schedule) can be defined as a partial order over the operations of a set of transactions.

\[ H_1 = \{W_2(x), R_1(x), R_3(x), W_1(x), C_1, W_2(y), R_3(y), R_2(z), C_2, R_3(z), C_3\} \]
Serial History

- All the actions of a transaction occur consecutively.
- No interleaving of transaction operations.
- If each transaction is consistent (obeys integrity rules), then the database is guaranteed to be consistent at the end of executing a serial history.

\[
\begin{align*}
T_1 &: \text{Read}(x) \quad T_2 &: \text{Write}(x) \quad T_3 &: \text{Read}(x) \\
& \quad \text{Write}(x) \quad \text{Write}(y) \quad \text{Read}(y) \\
& \quad \text{Commit} \quad \text{Read}(z) \quad \text{Read}(z) \\
& \quad \text{Commit} \quad \text{Commit} \\
H &= \{W_2(x), W_2(y), R_2(z), R_1(x), W_1(x), R_3(x), R_4(y), R_3(z)\}
\end{align*}
\]

Serializable History

- Transactions execute concurrently, but the net effect of the resulting history upon the database is equivalent to some serial history.
- Equivalent with respect to what?
  - \textit{Conflict equivalence}: the relative order of execution of the conflicting operations belonging to unaborted transactions in two histories are the same.
  - \textit{Conflicting operations}: two incompatible operations (e.g., Read and Write) conflict if they both access the same data item.
    - Incompatible operations of each transaction is assumed to conflict; do not change their execution orders.
    - If two operations from two different transactions conflict, the corresponding transactions are also said to conflict.
Serializable History

<table>
<thead>
<tr>
<th>$T_1$: Read(x)</th>
<th>$T_2$: Write(x)</th>
<th>$T_3$: Read(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write(x)</td>
<td>Write(y)</td>
<td>Read(y)</td>
</tr>
<tr>
<td>Commit</td>
<td>Read(z)</td>
<td>Read(z)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>

The following are not conflict equivalent

$H_s = [W_2(x), W_2(y), R_2(z), R_1(x), W_1(x), R_3(y), R_3(z)]$

$H_1 = [W_2(x), R_1(x), R_3(y), R_3(z), W_1(x), R_3(y), R_3(z)]$

The following are conflict equivalent; therefore $H_2$ is serializable.

$H_s = [W_2(x), W_2(y), R_2(z), R_1(x), W_1(x), R_3(y), R_3(z)]$

$H_2 = [W_2(x), R_1(x), W_1(x), R_3(y), R_2(z), R_3(z)]$

Serializability in Distributed DBMS

- Somewhat more involved. Two histories have to be considered:
  - local histories
  - global history

- For global transactions (i.e., global history) to be serializable, two conditions are necessary:
  - Each local history should be serializable.
  - Two conflicting operations should be in the same relative order in all of the local histories where they appear together.
Global Non-serializability

\[ T_1: \quad \text{Read}(x) \quad \text{Write}(x) \quad \text{Read}(y) \quad \text{Write}(y) \quad \text{Commit} \]

\[ T_2: \quad \text{Read}(x) \quad \text{y} \leftarrow y+100 \quad \text{Write}(y) \quad \text{Commit} \]

- \(x\) stored at Site 1, \(y\) stored at Site 2
- \(LH_1, LH_2\) are individually serializable (in fact serial), but the two transactions are not globally serializable.

\[ LH_1 = \{R_1(x), W_1(x), R_2(x)\} \]
\[ LH_2 = \{R_2(y), R_1(y), W_1(y)\} \]

Concurrency Control Algorithms

- Pessimistic
  - Two-Phase Locking-based (2PL)
    - Centralized (primary site) 2PL
    - Primary copy 2PL
    - Distributed 2PL
  - Timestamp Ordering (TO)
    - Basic TO
    - Multiversion TO
    - Conservative TO
  - Hybrid

- Optimistic
  - Locking-based
  - Timestamp ordering-based
Locking-Based Algorithms

- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager).
- Locks are either read lock (rl) [also called shared lock] or write lock (wl) [also called exclusive lock]
- Read locks and write locks conflict (because Read and Write operations are incompatible)
  \[ rl \quad \text{yes} \quad \text{no} \]
  \[ wl \quad \text{no} \quad \text{no} \]
- Locking works nicely to allow concurrent processing of transactions.

Two-Phase Locking (2PL)

1. A Transaction locks an object before using it.
2. When an object is locked by another transaction, the requesting transaction must wait.
3. When a transaction releases a lock, it may not request another lock.
**Strict 2PL**

Hold locks until the end.

- **BEGIN**
- **period of data item use**
- **END**
- **Transaction duration**
- **Obtain lock**
- **Release lock**

**Centralized 2PL**

- There is only one 2PL scheduler in the distributed system.
- Lock requests are issued to the central scheduler.

```
Data Processors at participating sites Coordinating TM Central Site LM
```

- **Lock Request**
- **Lock Granted**
- **Operation**
- **End of Operation**
- **Release Locks**
**Distributed 2PL**

- 2PL schedulers are placed at each site. Each scheduler handles lock requests for data at that site.

- A transaction may read any of the replicated copies of item \( x \), by obtaining a read lock on one of the copies of \( x \). Writing into \( x \) requires obtaining write locks for all copies of \( x \).

---

**Distributed 2PL Execution**

<table>
<thead>
<tr>
<th>Coordinating TM</th>
<th>Participating LMs</th>
<th>Participating DPs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lock Request</strong></td>
<td><strong>Operation</strong></td>
<td></td>
</tr>
<tr>
<td><strong>End of Operation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Release Locks</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Timestamp Ordering

1. Transaction \( T_i \) is assigned a globally unique timestamp \( ts(T_i) \).
2. Transaction manager attaches the timestamp to all operations issued by the transaction.
3. Each data item is assigned a write timestamp \( wts \) and a read timestamp \( rts \):
   - \( rts(x) = \) largest timestamp of any read on \( x \)
   - \( wts(x) = \) largest timestamp of any read on \( x \)
4. Conflicting operations are resolved by timestamp order.

**Basic T/O:**
- **for** \( R_i(x) \)
  - **if** \( ts(T_i) < wts(x) \)
  - **then** reject \( R_i(x) \)
  - **else** accept \( R_i(x) \)
- \( rts(x) \leftarrow ts(T_i) \)
- **for** \( W_i(x) \)
  - **if** \( ts(T_i) < wts(x) \) **and** \( ts(T_i) < rts(x) \)
  - **then** reject \( W_i(x) \)
  - **else** accept \( W_i(x) \)
  - \( wts(x) \leftarrow ts(T_i) \)

## Conservative Timestamp Ordering

- Basic timestamp ordering tries to execute an operation as soon as it receives it
  - progressive
  - too many restarts since there is no delaying
- Conservative timestamping delays each operation until there is an assurance that it will not be restarted
- Assurance?
  - No other operation with a smaller timestamp can arrive at the scheduler
  - Note that the delay may result in the formation of deadlocks
Multiversion Timestamp Ordering

- Do not modify the values in the database, create new values.
- A $R_i(x)$ is translated into a read on one version of $x$.
  - Find a version of $x$ (say $x_r$) such that $ts(x_r)$ is the largest timestamp less than $ts(T_i)$.
- A $W_i(x)$ is translated into $W_i(x_w)$ and accepted if the scheduler has not yet processed any $R_j(x_r)$ such that
  $$ts(T_i) < ts(x_r) < ts(T_j)$$

Optimistic Concurrency Control Algorithms

Pessimistic execution

- Validate
- Read
- Compute
- Write

Optimistic execution

- Read
- Compute
- Validate
- Write
Optimistic Concurrency Control Algorithms

- Transaction execution model: divide into subtransactions each of which execute at a site
  - $T_{ij}$: transaction $T_i$ that executes at site $j$
- Transactions run independently at each site until they reach the end of their read phases
- All subtransactions are assigned a timestamp at the end of their read phase
- Validation test performed during validation phase. If one fails, all rejected.

Optimistic CC Validation Test

1. If all transactions $T_k$ where $ts(T_k) < ts(T_{ij})$ have completed their write phase before $T_{ij}$ has started its read phase, then validation succeeds
   - Transaction executions in serial order

```
T_k | R | V | W |
---|---|---|---|
T_{ij} | R | V | W |
```
Optimistic CC Validation Test

If there is any transaction $T_k$ such that $ts(T_k) < ts(T_{ij})$ and which completes its write phase while $T_{ij}$ is in its read phase, then validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \emptyset$.

- Read and write phases overlap, but $T_{ij}$ does not read data items written by $T_k$.

\[
\begin{array}{c|c|c|c|c|}
T_k & R & V & W \\
T_{ij} & R & V & W \\
\end{array}
\]

If there is any transaction $T_k$ such that $ts(T_k) < ts(T_{ij})$ and which completes its read phase before $T_{ij}$ completes its read phase, then validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \emptyset$ and $WS(T_k) \cap WS(T_{ij}) = \emptyset$.

- They overlap, but don't access any common data items.

\[
\begin{array}{c|c|c|c|c|}
T_k & R & V & W \\
T_{ij} & R & V & W \\
\end{array}
\]
Deadlock

- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking-based CC algorithms may cause deadlocks.
- TO-based algorithms that involve waiting may cause deadlocks.
- Wait-for graph
  - If transaction $T_i$ waits for another transaction $T_j$ to release a lock on an entity, then $T_i \rightarrow T_j$ in WFG.

Local versus Global WFG

Assume $T_1$ and $T_2$ run at site 1, $T_3$ and $T_4$ run at site 2. Also assume $T_3$ waits for a lock held by $T_4$ which waits for a lock held by $T_1$ which waits for a lock held by $T_2$ which, in turn, waits for a lock held by $T_3$. 
Deadlock Management

- **Prevention**
  - Guaranteeing that deadlocks can never occur in the first place. Check transaction when it is initiated. Requires no run time support.

- **Avoidance**
  - Detecting potential deadlocks in advance and taking action to insure that deadlock will not occur. Requires run time support.

- **Detection and Recovery**
  - Allowing deadlocks to form and then finding and breaking them. As in the avoidance scheme, this requires run time support.

Deadlock Prevention

- All resources which may be needed by a transaction must be predeclared.
  - The system must guarantee that none of the resources will be needed by an ongoing transaction.
  - Resources must only be reserved, but not necessarily allocated a priori.
  - Unsuitability of the scheme in database environment.
  - Suitable for systems that have no provisions for undoing processes.

- **Evaluation:**
  - Reduced concurrency due to preallocation.
  - Evaluating whether an allocation is safe leads to added overhead.
  - Difficult to determine (partial order).
  + No transaction rollback or restart is involved.
Deadlock Avoidance

- Transactions are not required to request resources a priori.
- Transactions are allowed to proceed unless a requested resource is unavailable.
- In case of conflict, transactions may be allowed to wait for a fixed time interval.
- Order either the data items or the sites and always request locks in that order.
- More attractive than prevention in a database environment.
- Wait-Die/Wound-Wait algorithms

Deadlock Detection

- Transactions are allowed to wait freely.
- Wait-for graphs and cycles.
- Topologies for deadlock detection algorithms
  - Centralized
  - Distributed
  - Hierarchical