Transaction Management in Distributed Database Systems

Transaction management in Oracle
Transaction Management in DDBS: What we will cover?

- Transactions and Their Properties
- Distributed Concurrency Control
- Concurrency Control Mechanisms
- Distributed Deadlock Management
- Transactions in Oracle
Recommended References

- Date C. J. (1990)
  - Fundamentals of Database Systems, 2nd Ed, Benjamin Cummings
- Korth HF & Silberschatz A (1991)
- Oszu,M.,Valduriez,P. Lecture Notes (Chapters 10 & 11)
- Oracle 10g Documentation
Outline

- Introduction
- Background
- Distributed DBMS Architecture
- Distributed Database Design
- Semantic Data Control
- Distributed Query Processing
  - Distributed Transaction Management
    - Transaction Concepts and Models
    - Distributed Concurrency Control
    - Distributed Reliability
  - Distributed Database Operating Systems
  - Open Systems and Interoperability
  - Parallel Database Systems
  - Distributed Object Management
  - Concluding Remarks
Transaction

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

- concurrency transparency
- failure transparency

Database in a consistent state  Database may be temporarily in an inconsistent state during execution  Database in a consistent state

Begin Transaction  Execution of Transaction  End Transaction
Transactions in … Communications

Each time you make a phone call, there is a call setup transaction that allocates some resources to your conversation; the call teardown is a second transaction, freeing those resources. The call setup increasingly involves complex algorithms to find the callee (800 numbers could be anywhere in the world) and to decide who is to be billed (800 and 900 numbers have complex billing). The system must deal with features like call forwarding, call waiting, and voice mail. After the call teardown, billing may involve many phone companies.

From J.Gray, TP Overview
Transactions in ... Finance

- When you purchase gas using a credit card, the point-of-sale terminal connects to the credit card company's computer. In case that fails, it may alternatively try to debit the amount to your account by connecting to your bank.

- This generalizes to all kinds of point-of-sale terminals such as cash registers, ATMs, etc.

- When banks balance their accounts with each other (electronic fund transfer), they use transactions for reliability and recoverability.

From J.Gray, TP Overview
Transactions in … Travel

- Making reservations for a trip requires many related bookings and ticket purchases from airlines, hotels, rental car companies, and so on.

- From the perspective of the customer, the whole trip package is one purchase. From the perspective of the multiple systems involved, many transactions are executed: One per airline reservation (at least), one for each hotel reservation, one for each car rental, one for each ticket to be printed, on for setting up the bill, etc.

- Along the way, each inquiry that may not have resulted in a reservation is a transaction, too.

From J.Gray, TP Overview
Order entry, job and inventory planning and scheduling, accounting, and so on are classical application areas of transaction processing. Computer integrated manufacturing (CIM) is a key technique for improving industrial productivity and efficiency. Just-in-time inventory control, automated warehouses, and robotic assembly lines each require a reliable data storage system to represent the factory state.

From J. Gray, TP Overview
Transactions in … Real-Time Systems

- This application area includes all kinds of physical machinery that needs to interact with the real world, either as a sensor, or as an actor. Traditionally, such systems were custom made for each individual plant, starting from the hardware. The usual reason for that was that 20 years ago off-the-shelf systems could not guarantee real-time behavior that is critical in these applications. This has changed, and so has the feasibility of building entire systems from scratch. Standard software is now used to ensure that the application will be portable.

From J.Gray, TP Overview
**Transaction Example – A Simple SQL Query**

**Transaction** BUDGET_UPDATE

begin

    EXEC SQL UPDATE J

    SET BUDGET = BUDGET * 1.1

    WHERE JNAME = "CAD/CAM"

end.
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{verbatim}
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")
endif
end. {Reservation}
\end{verbatim}
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

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)
  - \( RS \cup WS \)
Formalization

Let

1. \( O_{ij}(x) \) be some operation \( O_j \) of transaction \( T_i \) operating on entity \( x \), where \( O_j \in \{\text{read, write}\} \) and \( O_j \) is atomic

2. \( OS_i = \cup_j O_{ij} \)

3. \( N_i \in \{\text{abort, commit}\} \)

Transaction \( T_i \) is a partial order \( T_i = \{\Sigma_i, <_i\} \) where

1. \( \Sigma_i = OS_i \cup \{N_i\} \)

2. For any two operations \( O_{ij}, O_{ik} \in OS_i \), if \( O_{ij} = R(x) \) and \( O_{ik} = W(x) \) for any data item \( x \), then either

   \( O_{ij} <_i O_{ik} \) or \( O_{ik} <_i O_{ij} \)

3. \( \forall O_{ij} \in OS_i, O_{ij} <_i N_i \)
Example

Consider a transaction $T$:

- Read($x$)
- Read($y$)
- $x \leftarrow x + y$
- Write($x$)
- Commit

Then

$\Sigma = \{R(x), R(y), W(x), C\}$

$\leq \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$
Assume
\[ \leq \{ (R(x), W(x)), (R(y), W(x)), (R(x), C), (R(y), C), (W(x), C) \} \]
Properties of Transactions

**Atomicity**
- all or nothing

**Consistency**
- no violation of integrity constraints

**Isolation**
- concurrent changes invisible \(\Rightarrow\) serializable

**Durability**
- committed updates persist
Atomicity

- Either all or none of the transaction's operations are performed.
- Atomicity requires that if a transaction is interrupted by a failure, its partial results must be undone.
- The activity of preserving the transaction's atomicity in presence of transaction aborts due to input errors, system overloads, or deadlocks is called transaction recovery.
- The activity of ensuring atomicity in the presence of system crashes is called crash recovery.
Consistency

- Internal consistency
  - A transaction which executes *alone* against a *consistent* database leaves it in a consistent state.
  - Transactions do not violate database integrity constraints.

- Transactions are *correct* programs
Isolation

- **Serializable**
  - If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order.

- **Incomplete results**
  - An incomplete transaction cannot reveal its results to other transactions before its commitment.
  - Necessary to avoid cascading aborts.
Durability

- Once a transaction commits, the system must guarantee that the results of its operations will never be lost, in spite of subsequent failures.

- Database recovery
Characterization of Transactions

Based on

- Application areas
  - non-distributed vs. distributed
  - compensating transactions
  - heterogeneous transactions

- Timing
  - on-line (short-life) vs batch (long-life)

- Structure
  - flat (or simple) transactions
  - nested transactions

- Organization of read and write actions
  - two-step
  - restricted
  - action model
Transaction Structure

- Flat transaction
  - Consists of a sequence of primitive operations embraced between a begin and end markers.
    ```
    Begin_transaction Reservation
    ...
    end.
    ```

- Nested transaction
  - The operations of a transaction may themselves be transactions.
    ```
    Begin_transaction Reservation
    ...
    Begin_transaction Airline
    ...
    end. {Airline}
    Begin_transaction Hotel
    ...
    end. {Hotel}
    end. {Reservation}
    ```
Nested Transactions

- Have the same properties as their parents ⇒ may themselves have other nested transactions.
- Introduces concurrency control and recovery concepts to within the transaction.
- Types
  - Closed nesting
    - Subtransactions begin after their parents and finish before them.
    - Commitment of a subtransaction is conditional upon the commitment of the parent (commitment through the root).
  - Open nesting
    - Subtransactions can execute and commit independently.
    - Compensation may be necessary.
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

- Replica control protocols
  - How to control the mutual consistency of replicated data
  - One copy equivalence and ROWA
Architecture Revisited

- Begin_transaction, Read, Write, Commit, Abort
- Results
- Distributed Execution Monitor
- Transaction Manager (TM)
  - Scheduling/Descheduling Requests
- Scheduler (SC)
  - To data processor
- With other TMs
- With other SCs
Centralized Transaction Execution

User Application

Begin_Transaction, Read, Write, Abort, EOT

Transaction Manager (TM)

Read, Write, Abort, EOT

Results & User Notifications

Scheduler (SC)

Scheduled Operations

Recovery Manager (RM)

Results
Distributed Transaction Execution

User application

Begin_transaction, Read, Write, EOT, Abort

Results & User notifications

Read, Write, EOT, Abort

TM

SC

RM

Distributed Transaction Execution Model

Replica Control Protocol

Distributed Concurrency Control Protocol

Local Recovery Protocol

Distributed Database Management

© 1994 M. Tamer Özsu & Patrick Valduriez  Page 7. 25
Concurrence 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.
Execution Schedule (or History)

- An order in which the operations of a set of transactions are executed.
- A schedule (history) can be defined as a partial order over the operations of a set of transactions.

\[ T_1: \text{Read}(x) \quad T_2: \text{Write}(x) \quad T_3: \text{Read}(x) \]
\[ \quad \text{Write}(x) \quad \text{Write}(y) \quad \text{Read}(y) \]
\[ \quad \text{Commit} \quad \text{Read}(z) \quad \text{Read}(z) \]
\[ \quad \text{Commit} \quad \text{Commit} \]

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

A complete schedule $SC(T)$ over a set of transactions $T = \{T_1, \ldots, T_n\}$ is a partial order $SC(T) = \{\Sigma_T, <_T\}$ where

1. $\Sigma_T = \bigcup_i \Sigma_i$, for $i = 1, 2, \ldots, n$

2. $<_T \supseteq \bigcup_i <_i$, for $i = 1, 2, \ldots, n$

3. For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, either $O_{ij} <_T O_{kl}$ or $O_{kl} <_T O_{ij}$
Complete Schedule – Example

Given three transactions

- **T₁**: Read(x)  
  Write(x)  
  Commit
- **T₂**: Write(x)  
  Write(y)  
  Read(z)  
  Commit
- **T₃**: Read(x)  
  Read(y)  
  Read(z)  
  Commit

A possible complete schedule is given as the DAG

```
R₁(x) ← W₂(x) → R₃(x)
   |        |        |
   v        v        v
W₁(x) ← W₂(y) → R₃(y)
   |        |        |
   v        v        v
C₁    R₂(z) ← R₃(z)
   |        |        |
   v        v        v
C₂    C₃
```
Schedule Definition

A schedule is a prefix of a complete schedule such that only some of the operations and only some of the ordering relationships are included.

\[ T_1: \text{Read}(x) \quad T_2: \text{Write}(x) \quad T_3: \text{Read}(x) \]

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

\[ \text{Commit} \quad \text{Read}(z) \quad \text{Read}(z) \]

\[ \text{Commit} \]

\[ R_1(x) \xrightarrow{} W_2(x) \xrightarrow{} R_3(x) \]

\[ W_1(x) \xrightarrow{} W_2(y) \xrightarrow{} R_3(y) \]

\[ C_1 \xrightarrow{} R_2(z) \xrightarrow{} R_3(z) \]

\[ C_2 \xrightarrow{} C_3 \]

\[ R_1(x) \xrightarrow{} W_2(x) \xrightarrow{} R_3(x) \]

\[ W_2(y) \xrightarrow{} R_3(y) \]

\[ R_2(z) \xrightarrow{} R_3(z) \]
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.

\[
\begin{align*}
T_1: \quad & \text{Read}(x) \quad \text{Write}(x) \quad \text{Commit} \\
T_2: \quad & \text{Write}(y) \quad \text{Read}(y) \quad \text{Commit} \\
T_3: \quad & \text{Read}(x) \quad \text{Read}(z) \quad \text{Commit}
\end{align*}
\]

\[H_5 = \{W_2(x), W_2(y), R_2(z), C_2, R_1(x), W_1(x), C_1, R_3(x), R_3(y), R_3(z), C_3\}\]
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?
  - **Conflict equivalence**: the relative order of execution of the conflicting operations belonging to un aborted transactions in two histories are the same.
  - **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

\[ T_1: \text{Read}(x) \quad T_2: \text{Write}(x) \quad T_3: \text{Read}(x) \]

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

\[ \text{Commit} \quad \text{Read}(z) \quad \text{Read}(z) \]

\[ \text{Commit} \]

The following are not conflict equivalent

\[ H_2 = \{ W_2(x) , W_2(y) , R_2(z) , C_2 , R_1(x) , W_1(x) , C_1 , R_3(x) , R_3(y) , R_3(z) , C_3 \} \]

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

The following are conflict equivalent; therefore

\[ H_2 \text{ is } \textit{serializable}. \]

\[ H_2 = \{ W_2(x) , R_1(x) , W_1(x) , C_1 , R_3(x) , W_2(y) , R_3(y) , R_2(z) , C_2 , R_3(z) , C_3 \} \]
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

\[ T_1: \begin{align*}
&\text{Read}(x) \\
&x \leftarrow x + 5 \\
&\text{Write}(x) \\
&\text{Commit}
\end{align*} \quad \begin{align*}
T_2: \begin{align*}
&\text{Read}(x) \\
&x \leftarrow x \times 15 \\
&\text{Write}(x) \\
&\text{Commit}
\end{align*}
\]

The following two local histories are individually serializable (in fact serial), but the two transactions are not globally serializable.

\[ LH_1 = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\} \]

\[ LH_2 = \{R_2(x), W_2(x), C_2, R_1(x), W_1(x), C_1\} \]
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
Problems of Database Concurrency

- Interleaved executions of transactions to increase throughput may generate a number of problems if not controlled properly:
  - The Lost Update  P1
  - Reading Uncommitted Changes  P2
  - Unrepeatable Reads  P3
  - Phantom Problem  P3

- First we will look at what these problems are, then we will look at controls the DBMS may use to avoid these problems.
Database Concurrency: Lost Updates

Trans 1

Read P1 (20)
QOH = QOH + 15
Write P1 (35)

Trans 2

Read P1 (20)
QOH = QOH - 10
Write P1 (10)
Database Concurrency: Reading Uncommitted Changes

Trans 1

Read P1 (20)
QOH = QOH + 15
Write P1 (35)

Trans 1 Aborts

Trans 2

Read P1 (35)
QOH = QOH + 20
Write P1 (55)
Database Concurrency: Unrepeatable Reads

Trans 1

Database

Part # QOH
P1 20

Read P1 (20)
QOH = QOH + 15
Write P1 (35)

Trans 2

Read P1 (20)

Read P1 (35)
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)
  
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rl</td>
<td>wl</td>
</tr>
<tr>
<td>rl</td>
<td>yes</td>
</tr>
<tr>
<td>wl</td>
<td>no</td>
</tr>
</tbody>
</table>
- 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.

![Diagram showing two-phase locking]

- Lock point
- Obtain lock
- Release lock

Phase 1
Phase 2
Strict 2PL

Hold locks until the end.

- Obtain lock
- Release lock

No. of locks

BEGIN

END

period of data item use

Transaction duration
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

Operation → Lock Request → Lock Granted → Release Locks → End of Operation
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

Coordinating TM  Participating LMs  Participating DPs

Lock Request

Operation

End of Operation

Release Locks
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 ($rts$):
   - $rts(x) = \text{largest timestamp of any read on } x$
   - $wts(x) = \text{largest timestamp of any read on } x$
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 $ts(T_i) < rts(x) \text{ and } ts(T_i) < wts(x)$
       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 $x_v$) such that $ts(x_v)$ is the largest timestamp less than $ts(T_j)$.
- 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

2. 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 $T_k$
Optimistic CC Validation Test

3 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.
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 $T_3$.

**Local WFG**

Site 1

- $T_1$
- $T_2$

Site 2

- $T_4$
- $T_3$

**Global WFG**

- $T_1$
- $T_2$
- $T_3$
- $T_4$
Deadlock Management

- Ignore
  - Let the application programmer deal with it, or restart the system

- 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.
Deadlock Avoidance –
Wait-Die & Wound-Wait Algorithms

**WAIT-DIE Rule:** If $T_i$ requests a lock on a data item which is already locked by $T_j$, then $T_i$ is permitted to wait iff $ts(T_i) < ts(T_j)$. If $ts(T_i) > ts(T_j)$, then $T_i$ is aborted and restarted with the same timestamp.

- if $ts(T_i) < ts(T_j)$ then $T_i$ waits else $T_i$ dies
- non-preemptive: $T_i$ never preempts $T_j$
- prefers younger transactions

**WOUND-WAIT Rule:** If $T_i$ requests a lock on a data item which is already locked by $T_j$, then $T_i$ is permitted to wait iff $ts(T_i) > ts(T_j)$. If $ts(T_i) < ts(T_j)$, then $T_j$ is aborted and the lock is granted to $T_i$.

- if $ts(T_i) < ts(T_j)$ then $T_j$ is wounded else $T_i$ waits
- preemptive: $T_i$ preempts $T_j$ if it is younger
- prefers older transactions
Deadlock Detection

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

- One site is designated as the deadlock detector for the system. Each scheduler periodically sends its local WFG to the central site which merges them to a global WFG to determine cycles.

- How often to transmit?
  - Too often ⇒ higher communication cost but lower delays due to undetected deadlocks
  - Too late ⇒ higher delays due to deadlocks, but lower communication cost

- Would be a reasonable choice if the concurrency control algorithm is also centralized.

- Proposed for Distributed INGRES
Hierarchical Deadlock Detection

Build a hierarchy of detectors

```
   DD_{ox}
    /   \
   /     \
DD_{11} / \ DD_{14}
  /       \
Site 1  Site 2  Site 3  Site 4
```

DD_{21} DD_{22} DD_{23} DD_{24}
Distributed Deadlock Detection

- Sites cooperate in detection of deadlocks.
- One example:
  - The local WFGs are formed at each site and passed on to other sites. Each local WFG is modified as follows:
    1. Since each site receives the potential deadlock cycles from other sites, these edges are added to the local WFGs
    2. The edges in the local WFG which show that local transactions are waiting for transactions at other sites are joined with edges in the local WFGs which show that remote transactions are waiting for local ones.
  - Each local deadlock detector:
    - looks for a cycle that does not involve the external edge. If it exists, there is a local deadlock which can be handled locally.
    - looks for a cycle involving the external edge. If it exists, it indicates a potential global deadlock. Pass on the information to the next site.
A transaction is a logical unit of work that contains one or more SQL statements.

A transaction begins with the first executable SQL statement. A transaction ends when it is committed or rolled back, either explicitly (with a COMMIT or ROLLBACK statement) or implicitly (when a DDL statement is issued).
Statement Execution and Transaction Control

- An SQL statement that "executes successfully" is different from a "committed" transaction.

- Executing successfully means that a single statement was parsed and found to be a valid SQL construction, and that the entire statement executed without error as an atomic unit (for example, all rows of a multirow update are changed). However, until the transaction that contains the statement is committed, the transaction can be rolled back, and all of the changes of the statement can be undone. A statement, rather than a transaction, executes successfully.

- Committing means that a user has said either explicitly or implicitly "make the changes in this transaction permanent". The changes made by the SQL statement(s) of your transaction become permanent and visible to other users only after your transaction has been committed. Only other users' transactions that started after yours will see the committed changes.
Statement-Level Rollback

- If at any time during execution an SQL statement causes an error, all effects of the statement are rolled back. The effect of the rollback is as if that statement were never executed. This is a statement-level rollback.

- Errors discovered during SQL statement execution cause statement-level rollbacks. (An example of such an error is attempting to insert a duplicate value in a primary key.) Errors discovered during SQL statement parsing (such as a syntax error) have not yet been executed, so do not cause a statement-level rollback. Single SQL statements involved in a deadlock (competition for the same data) may also cause a statement-level rollback.

- An SQL statement that fails causes the loss only of any work it would have performed itself; it does not cause the loss of any work that preceded it in the current transaction. If the statement is a DDL statement, the implicit commit that immediately preceded it is not undone.
Oralce and Transaction Management

- A transaction in Oracle begins when the first executable SQL statement is encountered. An executable SQL statement is an SQL statement that generates calls to an instance, including DML and DDL statements.

- When a transaction begins, Oracle assigns the transaction to an available rollback segment to record the rollback entries for the new transaction.

- A transaction ends when any of the following occurs:
  - You issue a COMMIT or ROLLBACK (without a SAVEPOINT clause) statement.
  - You execute a DDL statement (such as CREATE, DROP, RENAME, ALTER). If the current transaction contains any DML statements, Oracle first commits the transaction, and then executes and commits the DDL statement as a new, single statement transaction.
  - A user disconnects from Oracle. (The current transaction is committed.)
  - A user process terminates abnormally. (The current transaction is rolled back.)

- After one transaction ends, the next executable SQL statement automatically starts the following transaction.
Committing Transactions

Committing a transaction means making permanent the changes performed by the SQL statements within the transaction.

Before a transaction that has modified data is committed, the following will have occurred:
- Oracle has generated rollback segment records in rollback segment buffers of the system global area (SGA).
- The rollback information contains the old data values changed by the SQL statements of the transaction.
- Oracle has generated redo log entries in the redo log buffers of the SGA. These changes may go to disk before a transaction is committed.
- The changes have been made to the database buffers of the SGA. These changes may go to disk before a transaction actually is committed.

When a transaction is committed, the following occurs:
- The internal transaction table for the associated rollback segment records that the transaction has committed, and the corresponding unique system change number (SCN) of the transaction is assigned and recorded in the table.
- The log writer process (LGWR) writes the redo log entries in the redo log buffers of the SGA to the online redo log file. LGWR also writes the transaction's SCN to the online redo log file. This atomic event constitutes the commit of the transaction.
- Oracle releases locks held on rows and tables (see "Locking Mechanisms" on page 22-3 for a discussion of locks).
- Oracle marks the transaction "complete".
Rolling Back Transactions (1)

- Rolling back means undoing any changes to data that have been performed by SQL statements within an uncommitted transaction.
- Oracle allows you to roll back an entire uncommitted transaction. Alternatively, you can roll back the trailing portion of an uncommitted transaction to a marker called a savepoint.
- All types of rollbacks use the same procedures:
  - statement-level rollback (due to statement or deadlock execution error)
  - rollback to a savepoint
  - rollback of a transaction due to user request
  - rollback of a transaction due to abnormal process termination
  - rollback of all outstanding transactions when an instance terminates abnormally
  - rollback of incomplete transactions during recovery
Rolling Back Transactions (2)

- In rolling back an entire transaction, without referencing any savepoints, the following occurs:
  - Oracle undoes all changes made by all the SQL statements in the transaction by using the corresponding rollback segments.
  - Oracle releases all the transaction's locks of data.
  - The transaction ends.

- In rolling back a transaction to a savepoint, the following occurs:
  - Oracle rolls back only the statements executed after the savepoint.
  - The specified savepoint is preserved, but all savepoints that were established after the specified one are lost.
  - Oracle releases all table and row locks acquired since that savepoint, but retains all data locks acquired previous to the savepoint.
  - The transaction remains active and can be continued.
Savepoints

- You can declare intermediate markers called **savepoints** within the context of a transaction. Savepoints divide a long transaction into smaller parts.
- Using savepoints, you can arbitrarily mark your work at any point within a long transaction. You then have the option later of rolling back work performed before the current point in the transaction (the end of the transaction) but after a declared savepoint within the transaction.
- Savepoints are similarly useful in application programs in a similar way. If a procedure contains several functions, you can create a savepoint before each function begins. Then, if a function fails, it is easy to return the data to its state before the function began and then to reexecute the function with revised parameters or perform a recovery action.
- After a rollback to a savepoint, Oracle releases the data locks obtained by rolled back statements. Other transactions that were waiting for the previously locked resources can proceed. Other transactions that want to update previously locked rows can do so.
The Two-Phase Commit Mechanism

- In a distributed database, Oracle must coordinate transaction control over a network and maintain data consistency, even if a network or system failure occurs.

- A two-phase commit mechanism guarantees that all database servers participating in a distributed transaction either all commit or all roll back the statements in the transaction. A two-phase commit mechanism also protects implicit DML operations performed by integrity constraints, remote procedure calls, and triggers.

- The Oracle two-phase commit mechanism is completely transparent to users who issue distributed transactions.

- The recoverer (RECO) background process automatically resolves the outcome of *in-doubt distributed transactions*-distributed transactions in which the commit was interrupted by any type of system or network failure.
Discrete Transaction Management

- Application developers can improve the performance of short, nondistributed transactions by using the procedure BEGIN_DISCRETE_TRANSACTION. This procedure streamlines transaction processing so short transactions can execute more rapidly.

- During a discrete transaction, all changes made to any data are deferred until the transaction commits. Of course, other concurrent transactions are unable to see the uncommitted changes of a transaction whether the transaction is discrete or not.

- Oracle generates redo information, but stores it in a separate location in memory. When the transaction issues a commit request, Oracle writes the redo information to the redo log file (along with other group commits), and applies the changes to the database block directly to the block. Oracle returns control to the application once the commit completes. This eliminates the need to generate undo information, since the block actually is not modified until the transaction is committed, and the redo information is stored in the redo log buffers.
What’s next?

- Replication in distributed DB and Oracle