Transactions

- Many enterprises use databases to store information about their state
  - *e.g.*, Balances of all depositors at a bank

- When an event occurs in the real world that changes the state of the enterprise, a program is executed to change the database state in a corresponding way
  - *e.g.*, Bank balance must be updated when deposit is made

- Such a program is called a transaction
What does a Transaction do?

- **Update the database** to reflect the occurrence of a real world event
  - Deposit transaction: Update customer’s balance in database

- **Cause the occurrence of a real world event**
  - Withdraw transaction: Dispense cash (and update customer’s balance in database)

- **Return information from the database**
  - RequestBalance transaction: Outputs customer’s balance

---

A Sample Transaction

1: **Begin_Transaction**
2: get (K1, K2, CHF) from terminal
3: Select BALANCE Into S1 From ACCOUNT Where ACCOUNTNR = K1;
4: S1 := S1 - CHF;
5: Update ACCOUNT Set BALANCE = S1 Where ACCOUNTNR = K1;
6: Select BALANCE Into S2 From ACCOUNT Where ACCOUNTNR = K2;
7: S2 := S2 + CHF;
8: Update ACCOUNT Set BALANCE = S2 Where ACCOUNTNR = K2;
9: Insert Into BOOKING(ACCOUNTNR,DATE,AMOUNT,TEXT)
   Values (K1, today, -CHF, 'Transfer');
10: Insert Into BOOKING(ACCOUNTNR,DATE,AMOUNT,TEXT)
   Values (K2, today, CHF, 'Transfer');
12: If S1<0 Then **Abort_Transaction**
11: **End_Transaction**

Transaction = Program that takes database from one consistent state to another consistent state
So what is the issue?

- System crashes during transaction
  - database remains in inconsistent (intermediate) state
  - solution: recovery

- Multiple transactions executed at same time
  - other applications have access to inconsistent (intermediate) state
  - solution: concurrency control

DBMS View

- From the viewpoint of a DBMS, a transaction is a sequence of reads and writes that are supposed to make consistent transformations of system states while preserving system consistency

- Transaction is a “unit of work”, i.e. it must do all the work necessary to update the database and maintain integrity constraints.
Model for Transactions

- Assumption: the database is composed of *elements*
  - Usually 1 element = 1 block
  - Can be smaller (=1 record) or larger (=1 relation)

- Assumption: each transaction reads/writes some elements
Transaction operations

- A user’s program may carry out many operations on the data retrieved from DB but DBMS is only concerned about Read/Write.
- A database transaction is the execution of a program that include database access operations:
  - Begin-transaction
  - Read
  - Write
  - End-transaction
  - Commit-transaction
  - Abort-transaction
  - Undo
  - Redo

- Concurrent execution of user programs is essential for good DBMS performance.

State of a transaction

- **Active**: the transaction is executing.
- **Partially Committed**: the transaction ends after execution of final statement (“commit requested”).
- **Committed**: after successful completion checks.
- **Failed**: when normal execution can no longer proceed.
- **Aborted**: after the transaction has been rolled back.
Properties of a xAction

- The execution of each transaction must maintain the relationship between the database state and the enterprise state (at “all”times)

- Therefore additional requirements are placed on the execution of transactions beyond those placed on ordinary programs:
ACID Properties

- Atomicity
- Consistency
- Isolation
- Durability

**Atomicity** (all or nothing)
- A transaction is *atomic*: the effect of a transaction on the database should be either the effect of executing all its actions, or not executing any actions at all.

**Consistency** (no violation of integrity constraints)
- A transaction must preserve the consistency of a database after execution. (responsibility of the user)

**Isolation** (concurrent changes invisible -> serializable)
- Transaction is protected from the effects of concurrently executing other transactions.

**Durability** (committed updates persist)
- The effect of a committed transaction should persist even in the event of system failures such as a crash.
**Consistency**

- **Enterprise (Business) Rules** limit the occurrence of certain real-world events
  - Student cannot register for a course if the current number of registrants equals the maximum allowed

- Correspondingly, allowable database states are restricted, e.g., by
  - $\text{Current\_reg} \leq \text{max\_reg}$

- These limitations are called (static) **integrity constraints**: assertions that must be satisfied by the database state

---

**More on consistency**

- Other static consistency requirements are related to the fact that the database might store the same information in different ways
  - $\text{cur\_reg}=|\text{list\_of\_registered\_students}|$
  - Such limitations are also expressed as integrity constraints

- **Database is consistent** if all static integrity constraints are satisfied
More on Consistency

- A consistent database state does not necessarily model the actual state of the enterprise
  - A deposit transaction that increments the balance by the wrong amount maintains the integrity constraint $balance \geq 0$, but does not maintain the relation between the enterprise and database states

- A consistent transaction maintains database consistency and the correspondence between the database state and the enterprise state (implements its specification)
  - Specification of deposit transaction includes
    - $Balance = balance' + amt_{deposit}$
    - (where $balance'$ is the initial value of $balance$)

Dynamic Integrity Constraints

- Some constraints restrict allowable state transitions
  - A transaction might transform the database from one consistent state to another, but the transition might not be permissible
    - Example: A letter grade in a course (A, B, C, D, F) cannot be changed to an incomplete (I)

- Dynamic constraints cannot be checked by examining the database state
Transaction Consistency

- A **transaction is consistent** if, assuming the database is in a consistent state initially, when the transaction completes:
  1. All static integrity constraints are satisfied (constraints might have been violated in intermediate states)
     - Can be checked by examining a snapshot of the database
  2. The new state satisfies the specification of the transaction
     - Cannot be checked from a database snapshot
  3. No dynamic constraints have been violated
     - Cannot be checked from a database snapshot

Checking Constraints

- **Automatic**: Embed constraint in schema.
  - CHECK, ASSERTION for static constraints
  - TRIGGER for dynamic constraints
  - Increases confidence in correctness and decreases maintenance costs
  - Not always desirable since unnecessary checking (overhead) might result in performance degradation
    - Deposit transaction modifies *balance* but cannot violate constraint *balance ≥ 0*

- **Manual**: Perform check in application code.
  - Only necessary checks are performed
  - Scatters references to constraint throughout application
  - Difficult to maintain as transactions are modified/added
Atomicity

- A real-world event either happens or does not happen
  - Student either registers or does not register

- Similarly, the system must ensure that either the corresponding transaction runs to completion or, if not, it has no effect at all

- Not true of ordinary programs. A crash could leave files partially updated on recovery

Commit and Abort

- If the transaction successfully completes it is said to **commit**
  - The system is responsible for preserving the transaction’s results in spite of possible subsequent failures

- If the transaction does not successfully complete, it is said to **abort**
  - The system is responsible for undoing, or **rolling back**, any changes the transaction has made till the point of abort
Reasons for Aborting

- System crash

- Transaction aborted by system
  - Execution cannot be made atomic (e.g., if a site is down in a distributed transaction)
  - Execution did not maintain database consistency (integrity constraint is violated)
  - Execution was not isolated
  - Resources not available (deadlock)

- Transaction requests to roll back

API for Transactions

- DBMS and TP monitor provide commands for setting transaction boundaries. For example:
  - begin transaction
  - commit
  - rollback

- The commit command is a request
  - The system might commit the transaction, or it might abort it for one of the reasons on the previous slide

- The rollback command will always be executed
Durability

- The system must ensure that once a transaction commits, its effect on the database state is not lost in spite of subsequent failures.

- Not true of ordinary programs. A media failure after a program successfully terminates could cause the file system to be restored to a state that preceded the program’s execution.

More on Durability

- Database stored redundantly on mass storage devices.

- Architecture of mass storage devices affects type of media failures that can be tolerated:
  - **Availability**: extent to which a (possibly distributed) system can provide service despite failures.

- Non-stop DBMS (mirrored disks).

- Recovery based DBMS (log).
Isolation

- **Serial Execution**: Transactions execute one after the other
  - Each one starts after the previous one completes.
  - The execution of each transaction is isolated from all others.
  - If the initial database state and all transactions are consistent, all consistency constraints are satisfied and the final database state will accurately reflect the real-world state, *but*

- Serial execution is inadequate from a performance perspective

Schedules

- **Schedule** = an interleaving of actions (read/write) from a set of transactions, where the actions of any single transaction are in the original order

- **Complete Schedule** = add commit or abort at end

Initial State of DB + Schedule → Final State of DB
Serial Schedule

- One transaction at a time, no interleaving

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(acc1)</td>
<td>read(acc1)</td>
</tr>
<tr>
<td>acc1 := acc1 + 20</td>
<td>read(acc1)</td>
</tr>
<tr>
<td>write(acc1)</td>
<td>sum := sum + acc1</td>
</tr>
<tr>
<td>commit</td>
<td>write(sum)</td>
</tr>
<tr>
<td></td>
<td>commit</td>
</tr>
</tbody>
</table>

- Final state consistent (if transactions are)
- Different serial schedules give different final states

Isolation (2)

- **Concurrent execution** offers performance benefits:
  - A computer system has multiple resources capable of executing independently (*e.g.*, CPU’s, I/O devices), *but*
  - A transaction typically uses only one resource at a time
  - Concurrently executing transactions can make effective use of the system

- Concurrency is achieved by the DBMS by *interleaving* actions (reads/writes of DB objects) of various transactions.
Scheduling + CC

Arriving schedule (merge of transaction schedules)  
Schedule in which requests are serviced 

Concurrency Control 

To database 

Database server

Issues with Concurrent Scheduling

- Concurrent (interleaved) execution of a set of consistent transactions offers performance benefits, but might not be correct.

- Example: course registration; cur_reg is number of current registrants.

- Operations: read(Attribute: Value), write(Attribute: Value)

  \[ T1: r(cur\_reg:29) \quad w(cur\_reg:30) \]
  \[ T2: \quad r(cur\_reg:29) \quad w(cur\_reg:30) \]

- Result: Violation of static Integrity constraint of current_reg
Example

- Consider the following bank transactions:
  
  T1: Begin A=A+100, B=B-100 END
  T2: Begin A=1.06*A, B=1.06*B END

- Intuitively, the first transaction is transferring $100 from B’s account to A’s account. The second is crediting both accounts with a 6% interest payment.

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.

- However: The net effect should be equivalent to running these two transactions serially in some order.

Interleaving 1

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td></td>
<td>A=A+100</td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A=A×1.06</td>
</tr>
<tr>
<td>Read(B)</td>
<td>Write(A)</td>
</tr>
<tr>
<td>B=B-100</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B)</td>
</tr>
<tr>
<td></td>
<td>B=B×1.06</td>
</tr>
<tr>
<td></td>
<td>Write(B)</td>
</tr>
</tbody>
</table>

EFFECT: T1, T2
Interleaving 2

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>A=A*1.06</td>
<td>B=B*1.06</td>
</tr>
<tr>
<td>Write(A)</td>
<td>Write(B)</td>
</tr>
<tr>
<td>A=A+100</td>
<td>B=B-100</td>
</tr>
</tbody>
</table>

**EFFECT:** T2, T1

Interleaving 3

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>A=A+100</td>
<td>A=A*1.06</td>
</tr>
<tr>
<td>Write(A)</td>
<td>Write(A)</td>
</tr>
<tr>
<td></td>
<td>Read(A)</td>
</tr>
<tr>
<td></td>
<td>A=A*1.06</td>
</tr>
<tr>
<td></td>
<td>Write(A)</td>
</tr>
<tr>
<td></td>
<td>Read(B)</td>
</tr>
<tr>
<td></td>
<td>B=B*1.06</td>
</tr>
<tr>
<td></td>
<td>Write(B)</td>
</tr>
<tr>
<td></td>
<td>Read(B)</td>
</tr>
<tr>
<td></td>
<td>B=B-100</td>
</tr>
<tr>
<td></td>
<td>Write(B)</td>
</tr>
</tbody>
</table>

**PROBLEM!**

Interest for the same Rs 100 twice!
Interleaving 4

**PROBLEM!**

**Missing Interest!**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>Write(A)</td>
</tr>
<tr>
<td>A=A+100</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>A=A+100</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>B=B-100</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>B=B-100</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td>Write(B)</td>
</tr>
</tbody>
</table>

### Diagram

- **Problem:**
  - T1, T2: Read(A) → Write(A) → Read(A) → Write(A)
  - T2, T1: Read(A) → Write(A) → Read(A) → Write(A)

- **Missing Interest:**
  - T1, T2: Read(A) → Write(A) → Read(A) → Write(A)
  - T2, T1: Read(A) → Write(A) → Read(A) → Write(A)
Atomicity and Isolation

- Let
  $T_1: r(bal:10) w(bal, 1010)$ \hspace{2cm} abort
  $T_2: \hspace{2cm} r(bal, 1010) w(ok) \hspace{0.2cm} commit$

- $T_1$ deposits 1000
- $T_2$ grants credit and commits before $T_1$ completes
- $T_1$ aborts and rolls balance back to $10$
- $T_1$ has had an effect even though it aborted!

Concurrency Control

- Transforms arriving schedule into a correct interleaved schedule to be submitted to the DBMS
  - Delays servicing a request (reordering) -causes a transaction to wait
  - Refuses to service a request -causes transaction to abort

- Actions taken by concurrency control have performance costs
  - Goal is to avoid delaying servicing a request
Equivalence

- For interleaved schedules to be correct, they should be equivalent to serial schedules in their effect on the database for all applications.

- A strong notion of Equivalence (also called Conflict Equivalence) is based on the commutativity of operations.

- **Definition**: Database operations $p1$ and $p2$ commute if, for all initial database states, they return the same results and leave the database in the same final state when executed in either order.

Commutativity

- **Read**
  - $r(x, X)$-copy the value of database variable $x$ to local variable $X$

- **Write**
  - $w(x, X)$-copy the value of local variable $X$ to database variable $x$

- We use $r_T(x)$ and $w_T(x)$ to mean a read or write of $x$ by transaction $T1$
Commutativity

- $P1$ commutes with $P2$ if
  - They operate on different data items
    - $w1(x)$ commutes with $w2(y)$ and $r2(y)$
  - Both are reads
    - $r1(x)$ commutes with $r2(x)$

- Operations that do not commute **conflict**
  - $w1(x)$ conflicts with $w2(x)$
  - $w1(x)$ conflicts with $r2(x)$

Schedule Equivalence

- An interchange of adjacent operations of different transactions in a schedule creates an **equivalent schedule** if the operations commute
  - $S1$: $T_{11}$, $p_{ij}$, $p_{kl}$, $T_{12}$
  - $S2$: $T_{11}$, $p_{kl}$, $p_{ij}$, $T_{12}$ such that $i \neq k$

- Equivalence is **transitive**: If $S1$ is equivalent to $S2$ (by a series of such interchanges), and $S2$ is equivalent to $S3$, then $S1$ is equivalent to $S3$
Schedule Equivalence

- **Theorem** - Schedule S1 can be derived from S2 by a sequence of commutative interchanges if and only if conflicting operations in S1 and S2 are ordered in the same way.

  - **Only if**: Commutative interchanges do not reorder conflicting operations.
  
  - **If**: A sequence of commutative interchanges can be determined that takes S1 to S2 since conflicting operations do not have to be reordered.

- S1 and S5 are equivalent
- S5 is the serial schedule T1, T2
- S1 is NOT equivalent to the serial schedule T2, T1
Conflict Equivalence

- **Definition**: Two schedules, S1 and S2, of the same set of operations are *conflict equivalent* if conflicting operations are ordered in the same way in both.

- Or (using theorem) if one can be obtained from the other by a series of commutative interchanges.

Conflict Serializable

- **Definition**: A schedule is *conflict serializable* iff it is conflict equivalent to a serial schedule.

If in S transactions T1 and T2 have several pairs of conflicting operations (p_{1,1} conflicts with p_{2,1} and p_{1,2} conflicts with p_{2,2}) then p_{1,1} must precede p_{2,1} and p_{1,2} must precede p_{2,2} (or vice versa) in order for S to be serializable.
Serializable Schedules

- By default, “serializable” means “conflict serializable”

- Transactions are totally isolated in a serializable schedule

- A schedule is correct for any application if it is a serializable schedule of consistent transactions

- The schedule : $r_1(x)\ r_2(y)\ w_2(x)\ w_1(y)$ is not serializable
  - Why?

Intuition on Serializability

- Because T1 read x before T2 wrote it, T1 must precede T2 in any ordering, and because T1 wrote y after T2 read it, T1 must follow T2 in any ordering --- clearly an impossibility
Isolation

- An interleaved schedule of transactions is isolated if its effect is the same as if the transactions had executed serially in some order (serializable)

- Serializable schedules are always correct (if the single transactions are correct)

- Serializable is better than serial from a performance point of view

Serializability Graph

- Node for each transaction $T_i$

- Edge from $T_i$ to $T_j$ if there is an action of $T_i$ that precedes and “conflicts” with an action of $T_j$

- **Theorem**: A schedule is conflict serializable iff its Serializability Graph is acyclic.
Example

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{conflict}</td>
<td>\text{conflict}</td>
<td>\text{conflict}</td>
</tr>
<tr>
<td>\mathcal{W}_1(x)</td>
<td>\mathcal{R}_2(x)</td>
<td>\mathcal{W}_3(x)</td>
</tr>
<tr>
<td>\mathcal{R}_2(x)</td>
<td>\mathcal{W}_3(x)</td>
<td>\mathcal{R}_3(z)</td>
</tr>
<tr>
<td>\mathcal{W}_1(y)</td>
<td>\mathcal{R}_3(z)</td>
<td>\mathcal{W}_3(y)</td>
</tr>
<tr>
<td>\mathcal{W}_1(z)</td>
<td>\mathcal{W}_3(z)</td>
<td>\mathcal{W}_3(z)</td>
</tr>
</tbody>
</table>

Serializability graph

S is serializable in order \( T_1 \ T_2 \ T_3 \ T_4 \ T_5 \ T_6 \ T_7 \)

S is not serializable due to cycle \( T_2 \ T_6 \ T_7 \ T_2 \)
Recoverability: Schedules with Aborted Transactions

- T2 has aborted but has had an indirect effect on the database – schedule is unrecoverable

- **Problem:** T1 reads uncommitted data - dirty read

- **Solution:** A concurrency control is recoverable if it does not allow T1 to commit until all other transactions that wrote values T1 read have committed.

Cascaded Abort

Recoverable schedules solve abort problem but allow cascaded abort: abort of one transaction forces abort of another

- T1: r (y) w(z) abort
- T2: r (x) w(y) abort
- T3: w(x) abort

Better solution: prohibit dirty reads
Dirty Writes

*Dirty write:* A transaction writes a data item written by an active transaction

Dirty write complicates rollback:

\[
\text{no rollback necessary}
\]

\[
\begin{align*}
T_1 &: \text{w}(x) & \text{abort} \\
T_2 &: \text{w}(x) & \text{abort} \\
\end{align*}
\]

what value of \( x \) should be restored?

---

Strict Schedules

- *Strict schedule:* Dirty writes and dirty reads are prohibited
  - Strict and serializable are two different properties

- Strict, non-serializable schedule
  - \( \text{r1}(x) \ \text{w2}(x) \ \text{r2}(y) \ \text{w1}(y) \ \text{c1c2} \)

- Serializable, non-strict schedule
  - \( \text{w2}(x) \ \text{r1}(x) \ \text{w2}(y) \ \text{r1}(y) \ \text{c1c2} \)
Concurrency Control

- Concurrency control cannot see entire schedule:
  - It sees one request at a time and must decide whether to allow it to be serviced

- Strategy: Do not service a request if:
  - It violates strictness or serializability, or
  - There is a possibility that a subsequent arrival might cause a violation of serializability

![Diagram of concurrency control process]

Models of Concurrency Control

- **Immediate Update**
  - A write updates a database item
  - A read copies value from a database item
  - Commit makes updates durable
  - Abort undoes updates

- **Deferred Update**—*(we will likely not discuss this)*
  - A write stores new value in the transaction’s intentions list (does not update database)
  - A read copies value from database or transaction’s intentions list
  - Commit uses intentions list to durably update database
  - Abort discards intentions list
Models of Concurrency Control

- **Pessimistic**
  - A transaction requests permission for each database (read/write) operation
  - Concurrency control can:
    - *Grant* the operation (submit it for execution)
    - *Delay* it until a subsequent event occurs (commit or abort of another transaction), or
    - *Abort* the transaction
  - Decisions are made *conservatively* so that a commit request can *always* be granted
    - Takes precautions even if conflicts do not occur

Models of Concurrency Control

- **Optimistic**
  - Request for database operations (read/write) are *always* granted
  - Request to commit *might be denied*
  - Transaction is aborted if it performed a non serializable operation
  - Assumes that conflicts are not likely
    - The earlier it can aborted the better
Locking Implementation of an Immediate-Update Pessimistic Control

- A transaction can read a database item if it holds a read (shared) lock on the item

- It can read or update the item if it holds a write (exclusive) lock

- If the transaction does not already hold the required lock, a lock request is automatically made as part of the access

<table>
<thead>
<tr>
<th>Requested mode</th>
<th>Granted mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read</td>
</tr>
<tr>
<td>write</td>
<td>yes</td>
</tr>
</tbody>
</table>

Compatibility of locks

All locks held by a transaction are released when the transaction completes (commits or aborts)
Locking

- **Result**: A lock is not granted if the requested access conflicts with a prior access of an active transaction; instead the transaction waits. This enforces the rule:
  - Do not grant a request that imposes an ordering among active transactions (delay the requesting transaction)

- Resulting schedules are serializable and strict

Deadlocks

- **Problem**: Controls that cause transactions to wait can cause deadlocks
  - \( w1(x) \)  \( w2(y) \)  \( request_r1(y) \)  \( request_r2(x) \)

- **Solution**: Abort a transaction in the cycle
  - **Prevent Deadlock** - based on timestamps priorities
  - **Detect Deadlock** - by detecting a cycle in the wait-for graph when a request is delayed
  - **Time-Out** - Assume a deadlock when a transaction waits longer than some time-out period
Deadlock Prevention based on Timestamp Priorities

- Assign priorities based on timestamps (i.e., the older a transaction, the higher its priority).

- Assume Ti wants a lock that conflicts with a lock that Tj holds. Two policies are possible:
  - **Wait-Die**: If Ti has higher priority, Ti allowed to wait for Tj; otherwise (i.e., Ti younger): Ti aborts
  - **Wound-wait**: If Ti has higher priority, Tj aborts; otherwise (i.e., Ti younger): Ti waits

- If a transaction re-starts, make sure it has its original timestamp

---

Deadlock prevention based on timeouts

- A simple approach to deadlock resolution (pseudo prevention/detection) is based on lock request timeouts

- After requesting a lock on a locked data object, a transaction waits, but if the lock is not granted within a certain period, a deadlock is assumed and the waiting transaction is aborted and re-started.

- Very simple practical solution adopted by many DBMSs.
Wait-for Graphs – Deadlock Detection

- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock.
- Deadlock exists if there is a cycle in the graph.
- Periodically check for cycles in the waits-for graph.

<table>
<thead>
<tr>
<th>T1: S(A), R(A), S(B)</th>
<th>T2: X(B), W(B)</th>
<th>T3: S(C), R(C), X(A)</th>
<th>T4: X(B)</th>
</tr>
</thead>
</table>

Two Phase Locking

- Transaction does not release a lock until it has all the locks it will ever require.

- Transaction, T, has a locking phase followed by an unlocking phase

Two-phase locking (2PL) vs. Strict two-phase locking (strict 2PL)
Two Phase Locking

- A schedule produced by a two-phase locking control is:
  - Equivalent to a serial schedule in which transactions are ordered by the time of their first unlock operation
  - Not necessarily recoverable (dirty reads and writes are possible)

```
lock
T1: l(x) r(x) l(y) w(y) u(y) abort
```
```
unlock
T2: l(y) r(y) l(z) w(z) u(z) u(y) commit
```

Lock Granularity

- Data item: variable, record, row, table, file
- When an item is accessed, the DBMS locks an entity that contains the item. The size of that entity determines the granularity of the lock

1. **Coarse granularity** (large entities locked)
   - **Advantage**: If transactions tend to access multiple items in the same entity, fewer lock requests need to be processed and less lock storage space required
   - **Disadvantage**: Concurrency is reduced since some items are unnecessarily locked

2. **Fine granularity** (small entities locked)
   - Advantages and disadvantages are reversed
Granularity

- **Table locking** (*coarse*)
  - Lock entire table when a row is accessed.

- **Row (tuple) locking** (*fine*)
  - Lock only the row that is accessed.

- **Page locking** (compromise)
  - When a row is accessed, lock the containing page

Optimistic Concurrency Control

- No locking (and hence no waiting) means deadlocks are not possible

- Rollback is a problem if optimistic assumption is not valid: work of entire transaction is lost
  - With two-phase locking, rollback occurs only with deadlock
  - With optimistic concurrency control, rollback is only detected before transaction completes
Locking in RDBMS

- In the simple databases we have been studying, accesses are made to a named item, $x$, (for example $r(x)$), which can be locked.

- In relational databases, accesses are made to items that satisfy a predicate (for example, the set of rows returned by a SELECT statement)
  - What should we lock?
  - What is a conflict?

Operations on Accounts and Depositors conflict
Interleaved execution is not serializable
What do we lock?

- Lock tables:
  - Execution is serializable but ...
  - Performance suffers because lock granularity is coarse

- Lock rows:
  - Performance improves because lock granularity is fine but ...
  - Execution is not serializable

Problems with Row locking

- Audit
  1. Locks and reads Mary’s rows in Accounts

- NewAccount
  2. Inserts and locks new row, \( t \), in Accounts
  3. Locks and updates Mary’s row in Depositors
  4. Commits and releases all locks

- Audit
  5. Locks and reads Mary’s row in Depositors
Two SELECTs executed by Audit see inconsistent data
- The second sees effect of NewAccount; the first does not

Problem: Audit’s SELECT and NewAccount’s INSERT on Accounts do not commute, but the row locks held by Audit did not prevent NewAccount from INSERTing \( t \), (which satisfies the WHERE condition).
- \( t \) is referred to as a phantom

Phantoms
- Phantoms occur when row locking is used and
  - \( T_1 \) SELECTs, UPDATEs, or DELETEs using a predicate, \( P \)
  - \( T_2 \) creates a row (using INSERT or UPDATE) satisfying \( P \)

\[
T_1: \text{UPDATE Table} \quad \text{SET} \quad \text{Attr} = \ldots \quad \text{WHERE} \quad P \\
T_2: \text{INSERT INTO Table} \quad \text{VALUES} \quad (\ldots \text{satisfies } P\ldots)
\]
Preventing Phantoms

- Table locking does it; row locking does not
- **Predicate locking** does it
  - A predicate describes a set of rows, some are in a table and some are not; e.g. name = ‘Mary’
  - Every SQL statement has an associated predicate
  - When executing a statement, acquire a (read or write) lock on the associated predicate
  - Two predicate locks conflict if one is a write and there might be a row (not necessarily in the table) that is contained in both sets of tuples described by the predicates

### Example

- **Audit:**
  ```sql```
  SELECT SUM (balance) FROM Accounts WHERE name = ‘Mary’
  ```

- **NewAccount:**
  ```sql```
  INSERT INTO Accounts VALUES (‘123’, ‘Mary’, 100)
  ```

- Audit gets read lock on predicate name=’Mary’.
- NewAccount requests a write lock on predicate (acctnum=‘123’ ∧ name=’Mary’ ∧ bal=100)
- Request denied since predicates overlap
### Preventing Conflicts with Predicate Locks

| SELECT SUM (balance) FROM Accounts WHERE name = 'Mary' | DELETE FROM Accounts WHERE bal < 1 |

- Statements conflict since predicates overlap
- There might be an account with bal < 1 and name = ‘Mary’
- Locking is conservative: there might be no rows in Accounts satisfying both predicates

### Another Example

| SELECT SUM (balance) FROM Accounts WHERE name = 'Mary' | DELETE FROM Accounts WHERE name = 'John' |

- Statements commute since predicates are disjoint.
- There can be no rows (in or not in Accounts) that satisfy both predicates
Serializability in Relational DBs

- Predicate locking prevents phantoms and produces serializable schedules, but is very complex

- Table locking prevents phantoms and produces serializable schedules, but seriously affects performance

- Row locking does not prevent phantoms and can produce non-serializable schedules

- SQL defines several Isolation Levels weaker than SERIALIZABLE that allow non-serializable schedules and hence allow more concurrency

Weakening Serializability

- Weaker isolation levels:
  1. REPEATABLE READ,
  2. READ COMMITTED,
  3. READ UNCOMMITTED

- Increase performance by eliminating overhead and allowing higher degrees of concurrency
- Trade-off: sometimes you get the wrong answer
Example

CREATE TABLE Account
(accno INTEGER NOT NULL PRIMARY KEY,
name CHAR(30) NOT NULL,
balance FLOAT NOT NULL CHECK(balance >= 0));

Read Uncommitted

- Can read *dirty* data
- A data item is *dirty* if it is written by an uncommitted transaction
- Problem: What if the transaction that wrote the dirty data eventually aborts?
- Example: wrong average

```sql
-- T1:
UPDATE Account
SET balance = balance - 200
WHERE accno = 142857;
SELECT AVG(balance)
FROM Account;
ROLLBACK;

-- T2:
SELECT AVG(balance)
FROM Account;
COMMIT;
```
READ Committed

- No dirty reads, but non-repeatable reads possible
- Reading the same data item twice can produce different results
- Example: different averages

```
-- T1: -- T2:
SELECT AVG(balance) FROM Account;
UPDATE Account
SET balance = balance . 200
WHERE accno = 142857;
COMMIT;

SELECT AVG(balance) FROM Account;
COMMIT;
```

Repeatable Read

- Reads are repeatable, but may see *phantoms*
- Example: different average (still!)

```
T1: T2:
SELECT AVG(balance) FROM Account;
INSERT INTO Account
VALUES(428571, 1000);
COMMIT;

SELECT AVG(balance) FROM Account;
COMMIT
```
Isolation levels compared

<table>
<thead>
<tr>
<th>Isolation level/anomaly</th>
<th>Dirty reads</th>
<th>Non-repeatable reads</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Impossible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Impossible</td>
<td>Impossible</td>
<td>Possible</td>
</tr>
<tr>
<td>Serializable</td>
<td>Impossible</td>
<td>Impossible</td>
<td>Impossible</td>
</tr>
</tbody>
</table>

Summary

- Application programmer is responsible for creating consistent transactions
- DBMS and TP monitor are responsible for creating the abstractions of atomicity, durability, and isolation
- This greatly simplifies programmer’s task since he or she does not have to be concerned with failures or concurrency