Database Administration

- Transaction Processing
- Why Concurrency Control?
- Locking
- Database Recovery
- Query Optimization

Transactions

- Transaction -- A sequence of operations that is regarded as a single logical operation - i.e. a logical unit of work
- Atomicity of Transactions -- Either all operations in a transaction are executed completely or none of them is executed.
- Transaction Manager -- A component of a DBMS which is responsible for transaction processing.

Commit and Rollback of Transactions

```
TRANS: PROC OPTIONS (MAIN);
/* declarations omitted */
EXEC SQL WHENEVER SQLERROR GO TO UNDO;
GET LIST (SX, SY) /* get values from enduser */
EXEC SQL UPDATE SUPPLIER
   SET S# = :SY
   WHERE S# = :SX;
EXEC SQL UPDATE SHIPMENT
   SET S# = :SY
   WHERE S# = :SX;
EXEC SQL COMMIT;
GO TO FINISH
UNDO:   EXEC SQL ROLLBACK;
FINISH:   RETURN;
END /*TRANS */;
```
#### Transaction Processing

The execution of a transaction involves three steps:

- **Transaction Start**
  - Signals the beginning a transaction execution
  - Acquire required resources (e.g., private workspace)

- **Transaction Execution - Issues a sequence of reads/writes**
  - A read brings data from the DB to the private workspace
  - A write updates data in the private workspace and the DB

- **Transaction Commit/Rollback**
  - **Commit** – All changes become permanent in the DB
  - **Rollback** – Stops the execution and undos all changes

---

#### Synchronization Points

- A synchpoint represents a boundary between two consecutive transactions
- The point at which a database is guaranteed to be in a consistent state
- The only operations that establish a synchpoint are COMMIT, ROLLBACK, and program initiation
- COMMITS and ROLLBACKS terminate the transaction, not the program
  - All updates made by the program since the previous synchpoint are committed (COMMIT) or undone (ROLLBACK)
  - All database positioning is lost (all open cursors are closed)
  - All record locks are released

---

#### System Failure and Media Recovery

- **System failures** (e.g. power failure) - affect all transactions currently in progress, but do not physically damage the database
- **Media failure** (e.g. head crash on the disk) - cause damage to the database, or to some portion of it, and affect at least those transactions currently using that portion
- **System checkpoint** - Taking a checkpoint involves physically writing the contents of the database buffers out to the physical database, and b) physically writing a special checkpoint record out to the physical log
- **Undo a transaction** - any transaction that was in progress at the time of failure
- **Redo a transaction** - a transaction that was completed, but was not written to disk successfully before a system failure
System & Media Recovery (cont’d)

- A system failure has occurred at time $t_f$
- The most recent checkpoint prior to time $t_f$ was taken at time $t_c$
- Transactions of type $T_1$ were completed prior to time $t_c$
- Transaction of type $T_2$ started prior to time $t_c$ and completed after time $t_c$ and before time $t_f$
- Transactions of type $T_3$ also started prior to time $t_c$ but did not complete by time $t_f$
- Transactions of type $T_4$ started after time $t_c$ and completed before time $t_f$
- Transactions of type $T_5$ also started after time $t_c$ but did not complete by time $t_f$

System Recovery (cont’d)

1. Start with two lists of transactions, the UNDO-list and the REDO-list. Set the UNDO-list equal to the list of all transactions given in the checkpoint record; set the REDO-list to empty.
2. Search forward through the log, starting from the checkpoint record.
3. If a “start of transaction” log entry is found for transaction $T$, add $T$ to the UNDO-list.
4. If a “commit” log entry is found for transaction $T$, move $T$ from the UNDO-list to the REDO-list.
5. When the end of the log is reached, the UNDO- and REDO-lists identify, respectively, transactions of types $T_3$ and $T_5$, and transactions of types $T_2$ and $T_4$.
6. The system now works backward through the log, undoing transactions in the UNDO-list; then it works forward again, redoing trans. in the REDO-list.

Recovery Facilities

- Backup copies of the database
- Write-ahead logging (system log or journal)
  - Includes transaction IDs, time, before and after images of data record, etc.
- Checkpoint (synchronization point)
  - Synchronizes transaction execution and logs
  - Writes a checkpoint record to the log file
- Shadow copies
  - Create a shadow copy of entire data items (e.g., pages) for each update
  - Replace the originals upon the success of the transaction
  - Discarded upon the fail of the transaction
Database Recovery
-
- Media Failure Recovery
  - Restored by the last backup copy
  - Forward recovery if the system log is intact
-
- System Failure Recovery Using Logs
  - Undo (Backward Recovery).
    The uncommitted changes made by a transaction to a database are undone.
  - Redo (Forward Recovery).
    The committed changes made by a transaction are reapplied to an earlier copy of the database.

Concurrency Problems

Deposit and withdraw funds from a bank account.

**Transaction Start**
- Read account balance and store it in B;
- $B = B + \text{Deposit}\text{-Amount}$;
- Write B as the new balance;
- If ($B < \text{Withdraw}\text{-Amount}$) then
  - Transaction rollback;
- else
  - $B = B - \text{Withdraw}\text{-Amount}$;
  - Write B as the new balance;
  - Transaction commit;
- End.

Uncommitted Dependency Problem

Assume that two transactions T1 and T2 both access to a same account with $200 in balance. T1 withdraws $100. T2 deposits $50 first, and then withdraws $500.

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>read balance</td>
<td>start</td>
</tr>
<tr>
<td></td>
<td>balance = balance - $100</td>
<td>read balance</td>
</tr>
<tr>
<td></td>
<td>write balance</td>
<td>balance = balance + $50</td>
</tr>
<tr>
<td></td>
<td>commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>balance = balance - $500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>balance &lt; 500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rollback</td>
</tr>
</tbody>
</table>

What is the final balance?
### Lost Update Problem

Assume that two transactions T1 and T2 both access to a same account with $200 in balance. T1 reads a balance of $200 and withdraws $100. T2 reads a $200 balance and deposits 50.

- **Time**
  - **T1**
    - Start
    - Read balance
    - Balance = balance - $100
    - Write balance
    - Commit
  - **T2**
    - Start
    - Read balance
    - Balance = balance + $50
    - Write balance
    - Commit

What is the final balance?

### Inconsistent Analysis Problem

Assume three accounts: Acc 1 = Balance of $40, Acc 2 = Balance of $50, Acc 3 = Balance of $30

- **Time**
  - **T1**
    - Fetch Acc 1 (40)
    - Sum = 40
    - Fetch Acc 2 (50)
    - Sum = 90
    - Fetch Acc 3 (20)
    - Sum = 110, not 120
    - Fetch Acc 3 (30)
    - Update Acc 3
      - 30 -> 20
    - Update Acc 1
      - 40 -> 50
  - **T2**
    - Fetch Acc 3 (20)
    - Sum = 110, not 120

Can T1 write back to the database and will the DB be consistent?

### Concurrency Control Algorithms

- **Serializability** -- The execution of a set of transactions in some concurrent order is equivalent to the execution of those transactions in strictly sequential order.

- **Pessimistic concurrency control**
  - Locking
  - Timestamping
  - Multiple versions

- **Optimistic concurrency control**
Currency Control Using Locks

- **Two-Phase Locking**
  - Growing phase — Acquires required locks it will ever need.
  - Shrinking phase — Releases acquired locks and can't request any new lock

- **Read/Write Locks**
  - Read-lock (S) a.k.a. shared-lock — Non-exclusive
  - Write-lock (X) — Exclusive

Example Using R/W Locks

<table>
<thead>
<tr>
<th>T1</th>
<th>Time</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>start</td>
<td>start</td>
</tr>
<tr>
<td>write-lock (balance)</td>
<td>read balance</td>
<td>write-lock (balance)</td>
</tr>
<tr>
<td>waiting . . .</td>
<td>balance = balance + $50</td>
<td>balance &lt; 500</td>
</tr>
<tr>
<td>read balance</td>
<td>rollback (release lock)</td>
<td></td>
</tr>
<tr>
<td>balance = balance - $100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write balance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is the final balance?

Deadlocks

- **Deadlocks may occur when**
  - Transactions, T1 and T2, request locks in parallel:

<table>
<thead>
<tr>
<th>T1</th>
<th>Time</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>request X lock on R1</td>
<td></td>
<td>request X lock on R2</td>
</tr>
<tr>
<td>request X lock on R2</td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>wait</td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>wait</td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>wait</td>
<td></td>
<td>wait</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
<td>etc.</td>
</tr>
</tbody>
</table>
**What Is Query Optimization?**

- A process that translates a query into an equivalent form which is more efficient and costs less than the original one.
- Query optimization is responsible by the DBMS
- Supports associative queries

Example:

```sql
SELECT Student.id, Student.name
FROM Student, Class
WHERE Student.id = Class.id AND Class.course # = 'CIS3210';
```

Assume:
- 10,000 students
- 100 student are taking CIS3210
- each student takes 3 classes

---

**No Optimization Example**

<table>
<thead>
<tr>
<th>Student</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>30,000 tuples</td>
<td>30,000 tuples</td>
</tr>
<tr>
<td>Student.id=Class.id</td>
<td>course #='CIS3210'</td>
</tr>
<tr>
<td>Read: 3×10⁹ tuples</td>
<td>Write: 3×10⁹ tuples</td>
</tr>
</tbody>
</table>

**Optimization Example**

<table>
<thead>
<tr>
<th>Student</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>30,000 tuples</td>
<td>30,000 tuples</td>
</tr>
<tr>
<td>Student.id=Class.id</td>
<td>course #='CIS3210'</td>
</tr>
<tr>
<td>Read: 10⁶ tuples</td>
<td>Write: 10⁶ tuples</td>
</tr>
</tbody>
</table>
Translation of query into an internal representation

Relational algebra expressions

Find other equivalent internal representation

Several equivalent algebra expressions

Determine the execution cost of each representation

Choose a better representation and generate a query plan

---

Heuristical Processing Strategies

- Perform Projection as early as possible.
  - Keep projection attributes on same relation together.
- Perform Selection operations as early as possible.
  - Keep predicates on same relation together.
- Combine Cartesian product with subsequent Selection whose predicate represents join condition into a Join operation.
- Use associativity of binary operations to rearrange leaf nodes so that leaf nodes with most restrictive Selection operations executed first.
- Compute common expressions once.
  - If common expression appears more than once, and result not too large, store result and reuse it when required.
  - Useful when querying views, as same expression is used to construct view each time.

---

Cost Estimation for Attribute Operations

- Many different ways of implementing attribute operations.
- Aim of Query Optimization is to choose most efficient one.
- Use formulae that estimate costs for a number of options, and select one with lowest cost.
- Consider only cost of disk access, which is usually dominant cost in Query Processing.
- Many estimates are based on cardinality of the relation, so need to be able to estimate this.