Transactions and Concurrency Control

- Transaction: a unit of program execution that accesses and possibly updates some data items.

- A transaction is a collection of operations that logically form a single unit.

- Executing a transaction: either all operations are executed, or none are.

- Each transactions may consist of several steps, some involving I/O activity, and some CPU activity.

- Moreover, typically several transactions are running on a system; some are long, some are short.

- This creates an opportunity for concurrent execution. Problem: how to ensure consistency?
Transaction model

• Operation **read(X)** – transfers the data item X from the database to a local buffer belonging to the transaction

• Operation **write(X)** — transfers the data item X from the local buffer back to the database

• Example: transfer $100 from account A to account B
  
  `read(A);
  A := A - 100;
  write(A);
  read(A);
  read(B);
  B := B + 100;
  write(B)`

• It executes as a single unit: at no point between **read(A)** and **write(B)** can a user query the database, as it might be in an inconsistent state.
ACID properties

• **Atomicity**: either all operations of a transaction are reflected properly in the database, or none are.

• **Consistency**: execution of a transaction in isolation preserves the consistency of the database.

• **Isolation**: even though many transactions may run concurrently, the DBMS ensures that for any two transactions $T, T'$, it appears to $T$ that either $T'$ finished before $T$ started, or $T'$ started execution after $T$ finished.

• **Durability**: after a transaction completes successfully, the changes it has made persist.
States of a transaction

- **Active**: it stays in this state while it is executing
- **Partially committed**: after the final statement has been executed.
- **Failed**: after the discovery that normal execution cannot proceed.
- **Aborted**: after it has been rolled back, and the database state restored to the one prior to the start of the execution.
- **Committed**: after successful completion.
Two transactions

- $T$ takes $100$ from account $A$ to account $B$.
- $T'$ takes $10\%$ of account $A$ to account $B$.
- Property of $T$ and $T'$: they don't change $A+B$. Money isn't created, and doesn't disappear.

\[ T: \ \begin{align*}
\text{read}(A); \\
A & := A - 100; \\
\text{write}(A); \\
\text{read}(B); \\
B & := B+100; \\
\text{write}(B)
\end{align*} \]

\[ T': \ \begin{align*}
\text{read}(A); \\
tmp & := A*0.1; \\
A & := A - tmp; \\
\text{write}(A); \\
\text{read}(B); \\
B & := B+tmp; \\
\text{write}(B)
\end{align*} \]
Two serial executions: \( T;T' \) and \( T';T \)

<table>
<thead>
<tr>
<th>( T )</th>
<th>( T' )</th>
<th>( T' )</th>
<th>( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A);</td>
<td></td>
<td></td>
<td>read(A);</td>
</tr>
<tr>
<td>( A := A - 100; )</td>
<td>read(B);</td>
<td>( tmp := A*0.1; )</td>
<td>( A := A - 100; )</td>
</tr>
<tr>
<td>write(A);</td>
<td>( B = B+100; )</td>
<td>A := A - tmp;</td>
<td>( A := A - 100; )</td>
</tr>
<tr>
<td>read(B);</td>
<td>write(B);</td>
<td>write(A);</td>
<td>write(A);</td>
</tr>
<tr>
<td>( B = B+100; )</td>
<td></td>
<td>read(B);</td>
<td>read(B);</td>
</tr>
<tr>
<td>write(B);</td>
<td></td>
<td>( B = B+tmp; )</td>
<td>( B = B+100; )</td>
</tr>
</tbody>
</table>

Database Systems

L. Libkin
Transaction invariant

- A+B doesn’t change after $T$ and $T'$ execute.
- Assume that both A and B have $1,000.
- Evaluating $T; T'$:
  - after $T$: A=900, B=1100.
  - after $T'$: A=810, B=1190.
- Evaluating $T'; T$:
  - after $T'$: A=900, B=1100
  - after $T$: A=800, B=1200.
Concurrent execution I

\[ T \]
\begin{align*}
\text{read}(A); \\
A & := A - 100; \\
\text{write}(A);
\end{align*}

\[ T' \]
\begin{align*}
\text{read}(A); \\
tmp & := A \times 0.1; \\
A & := A - \text{tmp}; \\
\text{write}(A);
\end{align*}

\begin{align*}
\text{read}(B); \\
B & := B + 100; \\
\text{write}(B); \\
\text{read}(B); \\
B & := B + \text{tmp}; \\
\text{write}(B);
\end{align*}

Result:
\begin{align*}
A & = 810 \\
B & = 1190 \\
A + B & = 2000
\end{align*}
## Concurrent execution II

<table>
<thead>
<tr>
<th>$T$</th>
<th>$T'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A); A := A - 100; write(A);</td>
<td>read(A); tmp := A*0.1; A := A - tmp; write(A);</td>
</tr>
<tr>
<td>read(B); B := B + 100; write(B);</td>
<td>read(B); B := B + tmp; write(B);</td>
</tr>
</tbody>
</table>

Result:
A = 900
B = 1200
A + B = 2100
We created $100!$
Serializability

- Why is schedule I good and schedule II bad?
- Because schedule I is equivalent to a serial execution of $T$ and $T'$, and schedule II is not.
- We formalize this via conflict serializability.
- Transaction scheduling in DBMSs always ensures serializability.

Simplified representation of transactions

- For scheduling, the only important operations are read and write. What operations are performed on each data item does not affect the schedule.
- We thus represent transactions by a sequence of read-write operations, assuming that between each read(A) and write(A) some computation is done on A.
Simplified representation of transactions cont’d

- Examples of two concurrent executions in the new model:

<table>
<thead>
<tr>
<th>Schedule I</th>
<th>Schedule II</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>$T'$</td>
</tr>
<tr>
<td>read(A);</td>
<td>read(A);</td>
</tr>
<tr>
<td>write(A);</td>
<td>write(A);</td>
</tr>
<tr>
<td>read(A);</td>
<td>read(A);</td>
</tr>
<tr>
<td>write(A);</td>
<td>write(A);</td>
</tr>
<tr>
<td>read(B);</td>
<td>read(B);</td>
</tr>
<tr>
<td>write(B);</td>
<td>write(B);</td>
</tr>
<tr>
<td>read(B);</td>
<td>read(B);</td>
</tr>
<tr>
<td>write(B);</td>
<td>write(B);</td>
</tr>
<tr>
<td></td>
<td>read(B);</td>
</tr>
<tr>
<td></td>
<td>write(B);</td>
</tr>
</tbody>
</table>
Analyzing conflicts

• Let $\text{Op}_1$ and $\text{Op}_2$ be two consecutive operations in a schedule.

• Conflict – the order matters:

$$\text{Op}_1; \text{Op}_2 \quad \text{and} \quad \text{Op}_2; \text{Op}_1$$

may give us different results.

• If there is no conflict, $\text{Op}_1$ and $\text{Op}_2$ can be swapped.

• If $\text{Op}_1$ and $\text{Op}_2$ refer to different data items, they do not cause a conflict, and can be swapped.

• If they are both operations on the same data item $X$, then:

  if both are $\text{read}(X)$, the order does not matter;

  if $\text{Op}_1 = \text{read}(X)$, $\text{Op}_2 = \text{write}(X)$, the order matters.

  if $\text{Op}_1 = \text{write}(X)$, $\text{Op}_2 = \text{read}(X)$, the order matters.

  if $\text{Op}_1 = \text{write}(X)$, $\text{Op}_2 = \text{write}(X)$, the order matters.
Conflict serializability

- Swapping a pair of operations in a schedule is allowed when:
  - they refer to different data items, or,
  - they refer to the same data item and are both read operations.
- A schedule is called conflict serializable if it can be transformed into a serial schedule by a sequence of such swap operations.
Schedule I is conflict serializable

<table>
<thead>
<tr>
<th>$T$</th>
<th>$T'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A);</td>
<td></td>
</tr>
<tr>
<td>write(A);</td>
<td></td>
</tr>
<tr>
<td>read(B);</td>
<td></td>
</tr>
<tr>
<td>write(B);</td>
<td></td>
</tr>
</tbody>
</table>

$\rightarrow$

<table>
<thead>
<tr>
<th>$T$</th>
<th>$T'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A);</td>
<td></td>
</tr>
<tr>
<td>write(A);</td>
<td></td>
</tr>
<tr>
<td>read(B);</td>
<td></td>
</tr>
<tr>
<td>write(B);</td>
<td></td>
</tr>
</tbody>
</table>

$\rightarrow$

<table>
<thead>
<tr>
<th>$T$</th>
<th>$T'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A);</td>
<td></td>
</tr>
<tr>
<td>write(A);</td>
<td></td>
</tr>
<tr>
<td>read(B);</td>
<td></td>
</tr>
<tr>
<td>write(B);</td>
<td></td>
</tr>
</tbody>
</table>

$\rightarrow$

<table>
<thead>
<tr>
<th>$T$</th>
<th>$T'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A);</td>
<td></td>
</tr>
<tr>
<td>write(A);</td>
<td></td>
</tr>
<tr>
<td>read(B);</td>
<td></td>
</tr>
<tr>
<td>write(B);</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T$</th>
<th>$T'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A);</td>
<td></td>
</tr>
<tr>
<td>write(A);</td>
<td></td>
</tr>
<tr>
<td>read(B);</td>
<td></td>
</tr>
<tr>
<td>write(B);</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T$</th>
<th>$T'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A);</td>
<td></td>
</tr>
<tr>
<td>write(A);</td>
<td></td>
</tr>
<tr>
<td>read(B);</td>
<td></td>
</tr>
<tr>
<td>write(B);</td>
<td></td>
</tr>
</tbody>
</table>
Schedule II is not conflict serializable

\[
\begin{array}{l|l}
T & T' \\
\hline
read(A); & read(A); \\
write(A); & write(A); \\
read(B); & read(B); \\
write(B); & write(B);
\end{array}
\]

- In a serial schedule, either:
  - write(A) by \( T \) precedes read(A) by \( T' \), or
  - write(A) by \( T' \) precedes read(A) by \( T \)

- But:
  - write(A) by \( T \) cannot be swapped with write(A) by \( T' \), and
  - write(A) by \( T' \) cannot be swapped with read(A) by \( T \)

- Hence the schedule is not serializable.
Testing conflict serializability

- Construct the **precedence graph** of a schedule $S$
- Nodes: transactions in the system
- Edges: there is an edge $T \rightarrow T'$ if $T$ executes an operation $\text{Op}_1$ before $T'$ executes an operation $\text{Op}_2$ such that $\text{Op}_1$ and $\text{Op}_2$ cannot be swapped.

- That is, one of these conditions holds:
  - $T$ executes $\text{write}(X)$ before $T'$ executes $\text{read}(X)$
  - $T$ executes $\text{read}(X)$ before $T'$ executes $\text{write}(X)$
  - $T$ executes $\text{write}(X)$ before $T'$ executes $\text{write}(X)$
Testing conflict serializability

- Given a schedule $S$, construct its precedence graph.
- If there is an edge $T \rightarrow T'$, then in any serial schedule $S'$ equivalent to $S$, the transaction $T$ must appear before $T'$.
- A schedule $S$ is conflict serializable if and only if its precedence graph contains no cycles.
- Testing serializability:
  
  Construct the conflict graph
  
  Check if it has cycles
  
  If it doesn’t have cycles, do topological sort
- The result of the topological sort gives an equivalent serial schedule.
- Reminder: a topological sort of an acyclic graph $G$ produces an order $\prec$ on its nodes consistent with $G$ – if there is an edge from $x$ to $y$, then $x \prec y$. 
Testing conflict serializability: examples

• Thus, schedule I is conflict serializable, and schedule II is not.
Concurrency control: lock-based protocols

- Main goal of concurrency control: to ensure the isolation property for concurrently running transactions
- Typically achieved via locks
- Each data item is locked by at most one transaction; while it is locked, no other transaction has access to it.
- Two new primitives: lock(A) and unlock(A)
- A typical transaction:

  ```
  ...
  lock(A);
  read(A);
  ...
  write(A);
  unlock(A);
  ...
  ```
A new abstract view of transaction: only the order of lock and unlock operations matters.

If data item A is locked by transaction $T$, and transaction $T'$ issues a lock(A) command, it must wait until $T$ executes unlock(A).

New representation of schedules:

<table>
<thead>
<tr>
<th>Schedule $S_1$</th>
<th>Schedule $S_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_2$: lock(A)</td>
<td>$T_1$: lock(A)</td>
</tr>
<tr>
<td>$T_2$: unlock(A)</td>
<td>$T_2$: lock(B)</td>
</tr>
<tr>
<td>$T_3$: lock(A)</td>
<td>$T_2$: unlock(B)</td>
</tr>
<tr>
<td>$T_3$: unlock(A)</td>
<td>$T_1$: lock(B)</td>
</tr>
<tr>
<td>$T_1$: lock(B)</td>
<td>$T_1$: unlock(A)</td>
</tr>
<tr>
<td>$T_1$: unlock(B)</td>
<td>$T_2$: lock(A)</td>
</tr>
<tr>
<td>$T_2$: lock(B)</td>
<td>$T_2$: unlock(A)</td>
</tr>
<tr>
<td>$T_2$: unlock(B)</td>
<td>$T_1$: unlock(B)</td>
</tr>
</tbody>
</table>
Locking and serializability

- Precedence graph: for each operation $T_i$: unlock(A), locate the following $T_j$: lock(A) statement, and put an edge from $T_i$ to $T_j$

- Conflict-serializability with locking: a schedule is conflict-serializable if the precedence graph does not have cycles.

- Precedence graphs:

  - FOR S1
    - $S_1$ is conflict-serializable, $S_2$ is not.
A conflict-serializable schedule of **lock-unlock** operations ensures a conflict-serializable schedule of **read-write** operations:

\[
\begin{align*}
T_2 & \text{ lock(A)} & T_1 & \text{ } & T_2 & \text{ read(A)} \quad & \text{ write(A)} \quad & T_1 & \text{ read(B)} \quad & T_2 & \text{ write(B)} \\
T_2 & \text{ unlock(A)} & T_1 & \text{ read(B)} \quad & T_2 & \text{ write(A)} \\
T_1 & \text{ lock(B)} & T_1 & \text{ read(B)} \quad & T_2 & \text{ write(B)} \\
T_1 & \text{ unlock(B)} & T_1 & \text{ read(B)} \quad & T_2 & \text{ write(B)} \\
T_2 & \text{ lock(B)} & T_1 & \text{ read(B)} \quad & T_2 & \text{ write(B)} \\
T_2 & \text{ unlock(B)} & T_1 & \text{ read(B)} \quad & T_2 & \text{ write(B)}
\end{align*}
\]
Two-phase locking

- A protocol which guarantees conflict-serializable schedule
- Used in most commercial DBMSs
- Each transaction has two phases:
  - *Growing phase*: a transaction may request new locks, but may not release any locks
  - *Shrinking phase*: a transaction may release locks, but may not request any locks
- That is, after transaction released a lock, it may not request any new locks
- Main property of two-phase locking:
  
  A schedule $S$ in which every transaction satisfies the two-phase locking protocol is conflict-serializable
Two-phase locking and serializability: proof

Assume that every transaction conforms to the two-phase locking protocol (or: is a 2PL transaction)

Assume \( S \) is not serializable, and the precedence graph contains a cycle:

\[
T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_k \rightarrow T_1
\]

Trace the cycle:

- \( T_1 \) locks and unlocks something
- \( T_2 \) locks and unlocks something
- \( \ldots \)
- \( T_k \) locks and unlocks something
- \( T_1 \) locks and unlocks something

Then \( T_1 \) locks some data item after it released a lock, and hence it is not 2PL.
Two-phase locking and serializability cont'd

- The result is best possible.
- For any non-2PL transaction $T$, there is a 2PL transaction $T'$ and a schedule $S$ for $T, T'$ that is not serializable.

**Idea:**

<table>
<thead>
<tr>
<th>$T$</th>
<th>$T'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(A)</td>
<td>lock(A)</td>
</tr>
<tr>
<td>unlock(A)</td>
<td>lock(B)</td>
</tr>
<tr>
<td>lock(B)</td>
<td>unlock(A)</td>
</tr>
<tr>
<td>unlock(B)</td>
<td>unlock(B)</td>
</tr>
</tbody>
</table>

Schedule:

- $T$ lock(A), unlock(A)
- $T'$ all operations
- $T$ lock(B), unlock(B)

- This schedule is not serializable.
2PL in practice

- Majority of commercial DBMSs implement some form of 2PL
- Rigorous 2PL: all locks must be held until the transaction commits
- Two types of locks:
  - Exclusive: the transaction can both read and write the data item
  - Shared: the transaction can only read the data item
- At most one transaction can possess an exclusive lock for a data item at any given time, but several transactions can possess a shared lock
- Strict 2PL: all exclusive locks taken by the transaction must be held until it commits
- Strict and rigorous 2PL are the most common concurrency control mechanisms
Deadlocks

<table>
<thead>
<tr>
<th>time</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lock(A)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>lock(B)</td>
</tr>
<tr>
<td>3</td>
<td>lock(B)</td>
<td>now $T_1$ waits for $T_2$ to unlock B</td>
</tr>
<tr>
<td>4</td>
<td>lock(A)</td>
<td>now $T_2$ waits for $T_1$ to unlock A</td>
</tr>
<tr>
<td>5</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Deadlock: $T_1$ waits for $T_2$, $T_2$ waits for $T_1$

- In general, there is a set of transactions $T_1, \ldots, T_k$ such that:
  - $T_1$ waits for $T_2$
  - $T_2$ waits for $T_3$
  - ...
  - $T_k$ waits for $T_1$
Dealing with deadlocks

• Two main mechanisms: prevention and detection
• Prevention: find a concurrency control mechanism which ensures that there is no “waits-for” cycle
• Simple deadlock prevention: each transaction locks all data items before it starts execution. Such a locking constitutes one step.
• Disadvantage: data utilization is very low
• Another approach: use preemption. If $T_1$ has a lock on A, and $T_2$ requests it, then the system has three choices:
  - let $T_2$ wait, or
  - roll back $T_1$ and grant the lock to $T_2$, or
  - roll back $T_2$
Deadlock prevention cont’d

• The decision is based on timestamps, that say how old transactions are.

• Timestamp: the time when transaction started its execution. The larger the timestamp, the younger the transaction.

• Example: the wound-wait scheme. If $T_1$ requests a lock held by $T_2$, then $T_1$ waits if $T_1$ is younger than $T_2$. Otherwise $T_2$ is rolled back.

• The wait-die scheme. If $T_1$ requests a lock held by $T_2$, then $T_1$ waits if $T_1$ is older than $T_2$. Otherwise $T_1$ is rolled back.

• Issue: starvation may occur. Some transactions may never commit, as they keep being rolled back.
Deadlock detection and recovery

- The *wait-for* graph. Nodes are transactions. There is an edge from $T$ to $T'$ if $T'$ waits for $T$ to release a lock.
- There is a deadlock if there is a cycle in the wait-for graph.
- Deadlock recovery: identify a minimal set of transactions such that rolling them back will make the wait-for graph cycle-free.

```
T1  T2  T4
\arrow{T3} \arrow{T4}  \arrow{T1} \arrow{T3}
```

**DEADLOCK**  **AFTER DELETING T2**
Query Processing and Optimization

- *Query optimization*: finding a good way to evaluate a query
- Queries are declarative, and can be translated into procedural languages in more than one way
- Hence one has to choose the best (or at least good) procedural query
- This happens in the context of *query processing*
- A query processor turns queries and updates into sequences of operations on the database
Query processing and optimization stages

• Which relational algebra expression, equivalent to a given declarative query, will lead to the most efficient algorithm?

• For each algebraic operator, what algorithm do we use to compute that operator?

• How do operations pass data (main memory buffer, disk buffer?)

• We first concentrate the first step: finding efficient relational algebra expressions

• For the second step, we need to know how data is stored, and how it is accessed
Overview of query processing

- Start with a declarative query:

```sql
SELECT R.A, S.B, T.E
FROM R, S, T
```

- Translate into an algebra expression:

\[
\pi_{R.A, S.B, T.E}(\sigma_{R.A>5 \land S.B<3 \land T.D=T.E}(R \bowtie S \bowtie T))
\]

- Optimization step: rewrite to an equivalent but more efficient expression:

\[
\pi_{R.A, S.B, T.E}(\sigma_{A>5}(R) \bowtie \sigma_{B<3}(S) \bowtie \sigma_{D=E}(T))
\]

- Why is it more efficient?
  Because selections are evaluated early, and joined relations are not as large as \(R, S, T\).
Overview of query processing cont’d

• Evaluating the optimized expression. Choices to make: order of joins.
• Two possible query plans:

```
R S T
A>5 B<3 D=E
A,B
```

first joins $S$, $T$, and then joins the result with $R$. 
Overview of query processing cont’d

• Another query plan:

It first joins $S$, $T$, and then joins the result with $R$.

• Both query plans produce the same result.

• How to choose one?
Optimization by algebraic manipulations

• Given a relational algebra expression \( e \), find another expression \( e' \) equivalent to \( e \) that is easier (faster) to evaluate.

• Basic question: Given two relational algebra expressions \( e_1, e_2 \), are they equivalent?

• This is the same as asking if an expression \( e \) always produces the empty answer:

\[
e_1 = e_2 \iff e_1 - e_2 = \emptyset \text{ and } e_2 - e_1 = \emptyset
\]

• Problem: testing \( e = \emptyset \) is undecidable for relational algebra expressions.

• Good news:
  
  We can still list some useful equalities, and
  
  It is decidable for very important classes of queries (SPJ queries)
Optimization by algebraic manipulations

- Join and Cartesian product are commutative and associative, hence they can be applied in any order:

\[ R \times S = S \times R \]
\[ R \times (S \times T) = (R \times S) \times T \]
\[ R \bigotimes S = S \bigotimes R \]
\[ R \bigotimes (S \bigotimes T) = (R \bigotimes S) \bigotimes T \]

- Cascade of projections. Assume that attributes \( A_1, \ldots, A_n \) are among \( B_1, \ldots, B_m \). Then

\[ \pi_{A_1,\ldots,A_n}(\pi_{B_1,\ldots,B_m}(E)) = \pi_{A_1,\ldots,A_n}(E) \]

- Cascade of selections:

\[ \sigma_{c_1}(\sigma_{c_2}(E)) = \sigma_{c_1 \land c_2}(E) \]
Optimization by algebraic manipulations

- Commuting selections and projections. Assume that condition $c$ involves attributes $A_1, \ldots, A_n, B_1, \ldots, B_m$. Then

$$
\pi_{A_1, \ldots, A_n}(\sigma_c(E)) = \pi_{A_1, \ldots, A_n}(\sigma_c(\pi_{A_1, \ldots, A_n, B_1, \ldots, B_m}(E)))
$$

- A useful special case: if $c$ only involves attributes $A_1, \ldots, A_n$, then

$$
\pi_{A_1, \ldots, A_n}(\sigma_c(E)) = \sigma_c(\pi_{A_1, \ldots, A_n}(E))
$$

- Commuting selection with join. If $c$ only involves attributes from $E_1$, then

$$
\sigma_c(E_1 \Join E_2) = \sigma_c(E_1) \Join E_2
$$
Optimization by algebraic manipulations cont’d

• Let \( c_1 \) only mention attributes of \( E_1 \) and \( c_2 \) only mention attributes of \( E_2 \). Then

\[
\sigma_{c_1 \land c_2}(E_1 \Join E_2) = \sigma_{c_1}(E_1) \Join \sigma_{c_2}(E_2)
\]

• Because:

\[
\begin{align*}
\sigma_{c_1 \land c_2}(E_1 \Join E_2) & = \sigma_{c_1}(\sigma_{c_2}(E_1 \Join E_2)) \\
& = \sigma_{c_1}(E_1 \Join \sigma_{c_2}(E_2)) \\
& = \sigma_{c_1}(E_1) \Join \sigma_{c_2}(E_2)
\end{align*}
\]

• Another useful rule: If \( c \) only mentions attributes present in both \( E_1 \) and \( E_2 \), then

\[
\sigma_c(E_1 \Join E_2) = \sigma_c(E_1) \Join \sigma_c(E_2)
\]
Optimization by algebraic manipulations cont’d

• Rules combining $\sigma$, $\pi$ with $\cup$ and $-$

• Commuting selection and union:

$$\sigma_c(E_1 \cup E_2) = \sigma_c(E_1) \cup \sigma_c(E_2)$$

• Commuting selection and difference:

$$\sigma_c(E_1 - E_2) = \sigma_c(E_1) - \sigma_c(E_2)$$

• Commuting projection and union:

$$\pi_{A_1,\ldots,A_n}(E_1 \cup E_2) = \pi_{A_1,\ldots,A_n}(E_1) \cup \pi_{A_1,\ldots,A_n}(E_2)$$

• Question: what about projection and difference?

Is $\pi_A(E_1 - E_2)$ equal to $\pi_A(E_1) - \pi_A(E_2)$?
Optimization by algebraic manipulations: example

• Recall

\[ \pi_{R.A,S.B,T.E}(\sigma_{R.A>5} \land S.B<3 \land T.D=T.E(R \bowtie S \bowtie T)) \]

• Optimization: pushing selections

\[
\pi_{R.A,S.B,T.E}(\sigma_{R.A>5} \land S.B<3 \land T.D=T.E(R \bowtie S \bowtie T)) \\
= \pi_{R.A,S.B,T.E}(\sigma_{R.A>5}(\sigma_{S.B<3}(\sigma_{T.D=T.E(R \bowtie S \bowtie T))))) \\
= \pi_{R.A,S.B,T.E}(\sigma_{R.A>5}(\sigma_{S.B<3}(R \bowtie S \bowtie (\sigma_{T.D=T.E(T))]))) \\
= \pi_{R.A,S.B,T.E}(\sigma_{R.A>5}(R \bowtie \sigma_{S.B<3}(S) \bowtie (\sigma_{T.D=T.E(T))})) \\
= \pi_{R.A,S.B,T.E}(\sigma_{A>5}(R) \bowtie \sigma_{B<3}(S) \bowtie \sigma_{D=E(T)}))
\]
Implementation of individual operations

- Depends on access method and file organization
- Suppose EmplId is a key; how long does it take to answer:
  \[ \sigma_{EmplId=1234567}(Employee) \]?
- Time is linear in the worst case
- But one can perform the selection much faster if there is an \textit{index} on attribute EmplId
- \textbf{Index}: auxiliary structure that provides fast access to tuples in a table based on a given key value
- Most common example: B-trees
- With B-trees, the above selection takes \( O(\log n) \)
Note on indices and SQL

- SQL allows one to create an index on a given attribute or a sequence of attributes
- Once an index is created, the table can be accessed fast if the values of index attributes are known
- SQL always creates an index for attributes declared as a primary key of a table
- Syntax:

  CREATE INDEX <Index_Name> ON
  <Table_Name>(<attr1>,...,<attrN>)

- Example:

  CREATE INDEX EmplIndex ON Employee(EmplId)
Processing individual operators: join

- Join is the costliest operator of relational algebra
- Query: \[ R \Join S = \sigma_{R.A=S.A}(R \times S) \]
- SELECT R.A, R.B, S.C
  FROM R, S
  WHERE R.A=S.A

- Naive implementation:
  
  for every tuple t1 in R do
    for every tuple t2 in S do
      if t1.A=t2.A then output (t1.A,t1.B,t2.C)
    end
  end

- Time complexity: \( O(n^2) \)
Join processing

• Assumption: \( R.A \) is the primary key of \( R \).

• New \( O(n \log n) \) algorithm:

\[
\begin{align*}
\text{Sort } R \text{ and } S \text{ on attribute } A; \\
\text{scanS := first tuple in } S; \\
\text{for each tuple } t1 \text{ in } R \text{ do} \\
\quad \text{scan } S \text{ starting from scanS until a tuple} \\
\quad t2 \text{ with } t2.A \geq t1.A \text{ is found;} \\
\quad \text{if } t2.A = t1.A \text{ then} \\
\quad \quad \text{while } t2.A = t1.A \text{ do} \\
\quad \quad \quad \text{if } t1.A = t2.A \text{ then output } (t1.A, t1.B, t2.C); \\
\quad \quad \quad \text{move to the next tuple } t2 \text{ of } S \\
\quad \quad \text{end} \\
\quad \quad \text{set scanS := current tuple } t2 \\
\quad \text{end}
\end{align*}
\]
Join processing cont’d

• Previous algorithm can be extended to the case when the common attributes of $R$ and $S$ do not form a key in either relation

• One uses two pointers then to scan the relations

• Name: Sort-Merge join

• Both algorithms would be implemented differently in practice

• No need to do a new disk read to get each tuple; instead, read one block at a time

• Complexity of sort-merge join: If the relations are sorted, it requires $B_R + B_S$ disk reads, where $B_R, B_S$ are the numbers of disk blocks in $R, S$. 
Join processing: hash join

- Reminder: hashing.
- Bucket – a unit of storage that can store one or more tuples. Typically several disk blocks
- \( K \) – a set of search-key values; \( B \) – a set of buckets
- Hash function \( h : K \rightarrow B \)
- Good properties: uniform, random distribution
- Example of hashing: first two digits of the student # (assumes 100 buckets)
- Example of bad hashing: \( (\text{account balance mod 100 000}) \div 10 000 \)
- Overflows. Reason: bad distribution, insufficient buckets.
- Handling of overflows: a linked list of overflow buckets
Join processing: hash join of $R$ and $S$

- $X$ - the set of common attributes
- Step 1: Select $M$, the number of buckets
- Step 2: Select a hash function $h$ on attributes from $X$:
  \[ h: \{ \text{tuples over } X \} \rightarrow \{1, \ldots, M\} \]
- Step 3: Partition $R$ and $S$:

  \[
  \begin{align*}
  &\text{for each } t \text{ in } R \text{ do} \\
  &\quad i := h(t.X) \\
  &\quad H_i^R := H_i^R \cup \{t\} \\
  &\text{end}
  \end{align*}
  \]

  \[
  \begin{align*}
  &\text{for each } t \text{ in } S \text{ do} \\
  &\quad i := h(t.X) \\
  &\quad H_i^S := H_i^S \cup \{t\} \\
  &\text{end}
  \end{align*}
  \]

- If there are no overflows, this requires $O(B_R + B_S)$ I/O operations (read the relations, and write them back)
- With overflows, one uses recursive partitioning, and then complexity becomes $O(n \log n)$, where $n = B_R + B_S$. 
Join processing: hash join of $R$ and $S$

- Assume that the relations are partitioned
- Algorithm:

  for $i = 1$ to $M$ do
    read $H^R_i$
    read $H^S_i$
    add $H^R_i \bowtie H^S_i$ to the output
  end

- Why does it work? If two tuples $t_1 \in R$, $t_2 \in S$ match, $t_1.X = t_2.X$ and $h(t_1) = h(t_2)$; hence they are in the same partition class
- Improvements: how does one compute $H^R_i \bowtie H^S_i$? One possibility: use another hash function. If it doesn’t create overflows, the time for the algorithm is $O(B_R + B_S)$
Using hash functions for Boolean operations

• Observe:

\[ R \cap S = (H_1^R \cap H_1^S) \cup \ldots \cup (H_M^R \cap H_M^S) \]

• Because: if \( t \in R \cap S \) and \( t \in H_i^R \), then \( t \in H_i^S \)

• Advantage: each tuple \( t \in R \) must only be compared with \( H_{h(t)}^S \), and not with the whole relation \( S \)

• Using hash functions for difference:

\[
\text{for each } t \text{ in } R \text{ do}
\]
\[ i := h(t) \]
\[ \text{if } t \not\in H_i^S, \text{ include } t \text{ in the output} \]
\[ \text{end} \]
Other operations

• Set union $R \cup S$: if no index is needed on the result, just append $S$ to $R$

• If index is needed, then do as above, and then build a new index

• Duplicate elimination: On a sorted relation, it takes linear time. Thus, sort relation $R$ first, based on any attribute(s), and then do one pass and eliminate duplicate

• Complexity: $O(n \log n)$.

• Aggregation with GROUP BY: similarly, sort on the group by attributes, before computing aggregate functions.
Query processing cont’d

• Find names of theaters that play movies featuring Nicholson

```
SELECT S.theater
FROM Movies M, Schedule S
WHERE M.title=S.title AND M.actor='Nicholson'
```

• Translate into algebra:

```
π_{theater}(σ_{actor='Nicholson'}(M \bowtie S))
```

• Next step: choose a query plan

• To do so, use algebraic rewritings to create several equivalent expressions, and then choose algorithms for performing individual operators.
Query processing cont’d

Step 1

PLAN 1  PLAN 2  PLAN 3
Query processing cont’d

Step 2

(PLAN 1) (hash join)

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>S.theater</td>
</tr>
<tr>
<td>M</td>
<td>act='Nicholson'</td>
</tr>
</tbody>
</table>

(PLAN 2)

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>S.theater</td>
</tr>
<tr>
<td>M</td>
<td>act='Nicholson'</td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

(PLAN 3)

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>S.theater</td>
</tr>
<tr>
<td>M</td>
<td>act='Nicholson'</td>
</tr>
</tbody>
</table>

(PLAN 1) (sort-merge join, M.title is key)
Query processing cont’d

• Choosing the best plan: cost-based optimization
• Query optimizer estimates the cost of evaluating each plan
• Particularly important: selectivity estimation (how many tuples in $\sigma_c(E)$?)
  and join size estimation
• Techniques used: statistics. Sometimes a sampling is done before a query is processed.
• Problem with cost-based optimization: the set of all query plans is extremely large; the optimizer cannot try them all
• Another problem: how long can the optimizer run? Hopefully not as long as the savings it provides.
Join order

- We know that join is commutative and associative.
- How does one evaluate
  \[ R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 ? \]
- Possibilities:
  \[
  \begin{align*}
  & ( ( ( R_1 \bowtie R_2 ) \bowtie R_3 ) \bowtie R_4 ) \bowtie R_5 \\
  & (R_1 \bowtie R_2 ) \bowtie ( R_3 \bowtie ( R_4 \bowtie R_5 ) ) \\
  & ( R_1 \bowtie ( R_3 \bowtie R_5 ) ) \bowtie ( R_4 \bowtie R_2 )
  \end{align*}
  \]
- DB2 optimizer only considers deep join orders like the one on the left.
- In general, choosing an optimal join order is computationally hard (usually NP-complete for reasonable cost measures)
SQL and query optimization

- Query optimizer helps turn your query into a more efficient one, but you can help the query optimizer do its job better.

- The search space of all possible query plans is extremely large, and optimizers only run for a short time, and thus may fail to find a good plan.

- There are several rules that usually ensure a better query plan; however, a lot depends on a particular system, version, and its optimizers, and these rules may not be universally applicable. Still, if your query isn’t running fast enough, it’s worth giving them a try.
Order does matter!

SELECT * FROM Students WHERE grade='A' AND sex='female' is better than SELECT * FROM Students WHERE sex='female' AND grade='A'

- Because usually there are fewer A students than female students.
- Using orders instead of <>

SELECT * FROM Movies WHERE Length > 120 OR Length < 120 is better than SELECT * FROM Movies WHERE Length <> 120

- Because the ordered version forces SQL to use an index on Length, if there is one
- Without such an index, the version with OR runs longer
Provide more JOIN information

- SELECT *
  FROM T1, T2, T3
  WHERE T1.common = T3.common AND T1.common=T2.common

- SELECT *
  FROM T1, T2, T3
  WHERE T1.common = T3.common AND T3.common=T2.common

- These may not be as good as

  SELECT *
  FROM T1, T2, T3
  WHERE T1.common = T2.common
    AND T2.common = T3.common
    AND T3.common = T1.common
Avoid unions if OR is sufficient

```
SELECT *
FROM Personnel
WHERE location='Edinburgh'
UNION
SELECT *
FROM Personnel
WHERE location='Glasgow'
```

is not as good as

```
SELECT DISTINCT *
FROM Personnel
WHERE location='Edinburgh'
  OR location='Glasgow'
```

The optimizer normally works within a single SELECT-FROM-WHERE.
Joins are better than nested queries

```sql
SELECT S.Theater
FROM Schedule S
WHERE S.Title IN (SELECT M.Title
                    FROM Movies M
                    WHERE M.director='Spielberg')
```

is likely to run slower than

```sql
SELECT S.Theater
FROM Schedule S, Movies M
WHERE S.Title = M.Title
  AND M.director='Spielberg'
```