EECS 647: Introduction to Database Systems

Instructor: Luke Huan
Spring 2009
Query optimization

- One logical plan! “best” physical plan

- Questions
  - How to enumerate possible plans
  - How to estimate costs
  - How to pick the “best” one

- Often the goal is not getting the optimum plan, but instead avoiding the horrible ones

Any of these will do
Cost estimation

Physical plan example:

- We have: cost estimation for each operator
  - Example: \( \text{SORT}(CID) \) takes \( \log_2 B(input) \times B(input) \)
    - But what is \( B(input) \)?
- We need: size of intermediate results
Review DBMS Architecture

User/Web Forms/Applications/DBA

- Query Parser
- Query Rewriter
- Query Optimizer
- Query Executor
- Files & Access Methods
- Buffer Manager
- Storage Manager

Storage

- Transaction Manager
- Logging & Recovery
- Lock Manager

Buffers

Lock Tables

Main Memory
DBMS: a set of cooperating software modules
Transactions

- A program may carry out many operations on the data retrieved from the database.
- However, the DBMS is only concerned about what data is read/written from/to the database.
- **database** - a fixed set of relations \((A, B, C, \ldots)\)
- **transaction** - a sequence of read and write operations \((read(A), write(B), \ldots)\)
- DBMS’s abstract view of a user program
Correctness criteria: The ACID properties

- **A** tomicity: All actions in the Xact happen, or none happen.
- **C** onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one Xact is isolated from that of other Xacts.
- **D** urability: If a Xact commits, its effects persist.
An Example about SQL Transaction

Consider two transactions ($Xacts$):

- **T1**: BEGIN $A = A + 100$, $B = B - 100$ END
- **T2**: BEGIN $A = 1.06 \times A$, $B = 1.06 \times B$ END

- 1st xact transfers $100 from B’s account to A’s
- 2nd credits both accounts with 6% interest.
- Assume at first A and B each have $1000. What are the **legal outcomes** of running T1 and T2???
  - $1100 \times 1.06 = $1166

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the **net effect** *must* be equivalent to these two transactions running serially in some order.
Example (Contd.)

- Legal outcomes: A=1166, B=954 or A=1160, B=960
- Consider a possible interleaved schedule:

<table>
<thead>
<tr>
<th>T1:</th>
<th>A=A+100,</th>
<th>B=B-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>A=1.06*A,</td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

- This is OK (same as T1; T2). But what about:

<table>
<thead>
<tr>
<th>T1:</th>
<th>A=A+100,</th>
<th>B=B-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>A=1.06<em>A, B=1.06</em>B</td>
<td></td>
</tr>
</tbody>
</table>

- Result: A=1166, B=960; A+B = 2126, bank loses $6
- The DBMS’s view of the second schedule:

<table>
<thead>
<tr>
<th>T1:</th>
<th></th>
<th>R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A), W(A),</td>
<td>R(B), W(B)</td>
</tr>
</tbody>
</table>
SQL transactions

- Syntax in pgSQL:
  
  BEGIN
  <database operations>
  COMMIT [ROLLBACK]

- A transaction is automatically started when a user executes an SQL statement (begin is optional)

- Subsequent statements in the same session are executed as part of this transaction
  - Statements see changes made by earlier ones in the same transaction
  - Statements in other concurrently running transactions do not see these changes

- COMMIT command commits the transaction (flushing the update to disk)

- ROLLBACK command aborts the transaction (all effects are undone)
Atomicity

- Partial effects of a transaction must be undone when
  - User explicitly aborts the transaction using ROLLBACK
    - E.g., application asks for user confirmation in the last step and issues COMMIT or ROLLBACK depending on the response
  - The DBMS crashes before a transaction commits

- Partial effects of a modification statement must be undone when any constraint is violated
  - However, only this statement is rolled back; the transaction continues

- How is atomicity achieved?
  - Logging (to support undo)
Isolation

- Transactions must appear to be executed in a **serial schedule** (with no interleaving operations)
- For performance, DBMS executes transactions using a **serializable schedule**
  - In this schedule, only those operations that can be interleaved are executed concurrently
  - Those that can not be interleaved are in a serialized way
  - The schedule guarantees to produce the same effects as a serial schedule
Concurrent control

- Goal: ensure the “I” (isolation) in ACID

\[
\begin{align*}
T_1: & \quad T_2: \\
\text{read}(A); & \quad \text{read}(A); \\
\text{write}(A); & \quad \text{write}(A); \\
\text{read}(B); & \quad \text{read}(C); \\
\text{write}(B); & \quad \text{write}(C); \\
\text{commit}; & \quad \text{commit};
\end{align*}
\]
### Good versus bad schedules

<table>
<thead>
<tr>
<th>Good!</th>
<th>Bad!</th>
<th>Good! (But why?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>r$(A)$</td>
<td>r$(A)$</td>
<td>r$(A)$</td>
</tr>
<tr>
<td>w$(A)$</td>
<td>Read 400</td>
<td>w$(A)$</td>
</tr>
<tr>
<td>r$(B)$</td>
<td>Write $w(A)$</td>
<td>400 – 100</td>
</tr>
<tr>
<td>w$(B)$</td>
<td></td>
<td>w$(A)$</td>
</tr>
<tr>
<td>r$(A)$</td>
<td>r$(B)$</td>
<td>r$(B)$</td>
</tr>
<tr>
<td>w$(A)$</td>
<td></td>
<td>r$(C)$</td>
</tr>
<tr>
<td>r$(C)$</td>
<td>w$(B)$</td>
<td>w$(C)$</td>
</tr>
<tr>
<td>w$(C)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Good! Good! (But why?) Bad!

Read 400

Write $w(A)$

400 – 100

Write $w(A)$

400 – 50

r$(A)$

w$(A)$

r$(B)$

w$(A)$

r$(B)$

w$(A)$

r$(C)$

w$(B)$

w$(B)$

w$(C)$

w$(C)$
Serial schedule

- Execute transactions in order, with no interleaving of operations
  - $T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B), T_2.r(A), T_2.w(A)$
  - $T_2.r(C), T_2.w(C)$
  - $T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C), T_1.r(A), T_1.w(A)$
  - $T_1.r(B), T_1.w(B)$
  - Isolation achieved by definition!

- Problem: no concurrency at all

- Question: how to reorder operations to allow more concurrency
Conflicting operations

• Two operations on the same data item **conflict** if at least one of the operations is a write
  - r(X) and w(X) conflict
  - w(X) and r(X) conflict
  - w(X) and w(X) conflict
  - r(X) and r(X) do not
  - r/w(X) and r/w(Y) do not

• Order of conflicting operations matters
  - E.g., if $T_1.r(A)$ precedes $T_2.w(A)$, then conceptually, $T_1$ should precede $T_2$
Precedence graph

- A node for each transaction
- A directed edge from $T_i$ to $T_j$ if an operation of $T_i$ precedes and conflicts with an operation of $T_j$ in the schedule

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>w(B)</td>
</tr>
<tr>
<td>w(C)</td>
<td>w(C)</td>
</tr>
</tbody>
</table>

Good: no cycle

Bad: cycle
Conflict-serializable schedule

- A schedule is conflict-serializable iff its precedence graph has no cycles
- A conflict-serializable schedule is equivalent to some serial schedule (and therefore is “good”)
  - In that serial schedule, transactions are executed in the topological order of the precedence graph
  - You can get to that serial schedule by repeatedly swapping adjacent, non-conflicting operations from different transactions
Summary

- Transaction view of DBMS
  - Read(x)
  - Write(x)

- ACID
  - Atomicity: TX’s are either completely done or not done at all
  - Consistency: TX’s should leave the database in a consistent state
  - Isolation: TX’s must behave as if they are executed in isolation
  - Durability: Effects of committed TX’s are resilient against failures

- SQL transactions
  -- Begins implicitly
  SELECT ...;
  UPDATE ...;
  ROLLBACK | COMMIT;