Transactions and ACID properties

Introduction to Database Design 2011, Lecture 13
Course overview

- Communicating with DBMSs
- Designing databases
- Making databases efficient
- Making databases reliable
Overview

- Transactions
- ACID properties
- Concurrency and safe schedules
Transactions

• Transactions are units of work

• Example

  - Read balance of account A
  - Compute new balance = balance(A) - 50
  - Write new balance of A
  - Read balance of account B
  - Compute new balance = balance(B) + 50
  - Write balance of account B

• Transaction should be atomic: all or nothing
Transactions

- In database practice transactions are sequences of SQL statements
- Possibly intertwined by computations
- Can be written in
  - programming language (e.g. Java) accessing a database
  - procedural component of SQL
- SQL: `begin atomic ... end`
- Here we simplify and consider just sequences of reads and writes
ACID

- ACID stands for
  - Atomicity
  - Consistency
  - Isolation
  - Durability
Consistency

- Database must always be consistent wrt real world rules, e.g.,
  - Integrity constraints such as referential integrity must be satisfied
  - Rules of real world situation must be satisfied, e.g.,
    - Account balance must always be above a certain number (e.g. 0)
- Should also reflect real world as it is now
  - e.g. balance stored should correspond to actual balance of account
Transactional consistency

- Transaction leaves database in consistent state
  - (may assume database consistent before transaction start)
- May be required to satisfy other rules, e.g. leave the sum of the balances unchanged
  - (no money created or lost)
- During transaction consistency requirement may be violated temporarily
- Transactional consistency responsibility of transaction designer
Atomicity

- Transaction can be considered a unit of work
  - All or nothing!
- Consistency is impossible without atomicity
- Sometimes not possible to complete a started transaction, e.g.
  - In case of hardware failure or loss of connection
  - The application program may choose to abandon transaction
  - The DBMS may refuse to complete the transaction
- In these cases we say that transaction **fails**
Atomicity

• If a transaction fails it must be aborted
• This involves rolling back the transaction
• i.e., undoing all changes made by transaction
• Concurrency makes this complicated
  - e.g., changes made by transaction may have already been read
• Ability to roll back is implemented using a log of changes made to the database
Durability

• Durability is about trustworthy storage

• A transaction that has successfully completed is said to be **committed**

• Changes made by a committed transaction must be durable, i.e., able to survive
  - Power failure
  - Hardware failure etc
Durability

- When committing, changes must be written to non-volatile storage
- In practice, log is written to non-volatile storage
- Storage must be able to survive hardware failure
  - Maintain multiple copies of data
  - RAID
Transaction model

- Active
- Partially committed
  - Committed
  - Failed
  - Aborted
Isolation

• Transactions may not interfere with each other
• In reality transactions are executed concurrently
• Statements of transactions intertwined
• DBMS should create illusion of transactions being executed sequentially
• Isolation is necessary for consistency
Need for concurrency

- Resources operate in parallel
  - Multiple CPUs
  - Data stored on multiple disks

- Computation may be stalled while waiting for data

- Gains of concurrency
  - Increased throughput: idle resources can be utilised
  - Decreased waiting time: new transactions can start executing immediately
Concurrency and safe schedules
## A serial schedule

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A := A - 50</td>
<td>A := A + 20</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>B := B + 50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
</tbody>
</table>
A good concurrent schedule

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td>$A := A + 20$</td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td></td>
</tr>
<tr>
<td>$B := B + 50$</td>
<td></td>
</tr>
<tr>
<td>write($B$)</td>
<td></td>
</tr>
</tbody>
</table>
A bad concurrent schedule

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td>$A := A + 20$</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>$B := B + 50$</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
</tbody>
</table>
Schedules

• The DBMS receives a sequence of read and write requests from different transactions

• A schedule is an ordering of the reads and writes respecting the internal ordering in each transaction
Examples

- **Two schedules**

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td>write(B)</td>
<td></td>
</tr>
</tbody>
</table>

- **Schedule on right is equivalent to first $T_1$ then $T_2$**

- **It is up to DBMS to avoid bad schedules such as the one on the left**
Conflicting operations

- Two operations **commute** if the order in which they are executed does not affect the result

  \[
  \begin{align*}
  \text{read}(A), \text{read}(B) &= \text{read}(B), \text{read}(A) \\
  \text{write}(A), \text{read}(B) &= \text{read}(B), \text{write}(A) \\
  \text{write}(A), \text{write}(B) &= \text{write}(B), \text{write}(A) \\
  \text{read}(A), \text{read}(A) &= \text{read}(A), \text{read}(A)
  \end{align*}
  \]

- If they do not commute we say that they **conflict**

  \[
  \begin{align*}
  \text{write}(A), \text{read}(A) &\neq \text{read}(A), \text{write}(A) \\
  \text{write}(A), \text{write}(A) &\neq \text{write}(A), \text{write}(A)
  \end{align*}
  \]
Conflict equivalence

- Two schedules are **conflict equivalent** if they differ only up to swapping commuting operations.

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{read}(A) & \text{read}(A) \\
\text{write}(A) & \text{write}(A) \\
\text{read}(B) & \text{read}(B) \\
\text{write}(B) & \text{write}(B) \\
\end{array}
\]

- Executing conflict equivalent schedules gives same result.

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{read}(A) & \text{read}(A) \\
\text{write}(A) & \text{write}(A) \\
\text{read}(B) & \text{read}(B) \\
\text{write}(B) & \text{write}(B) \\
\end{array}
\]
A non-example

• The following schedules are **not** conflict equivalent

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

• Suppose e.g. $T_1$ transfers all money available in A to B
Conflict serializability

- A schedule is **conflict serializable** if it is conflict equivalent to a serial schedule

- Example:

```
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<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></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
</tbody>
</table>
```
A non-serializable schedule

- The following is neither conflict equivalent to $T_1T_2$ nor $T_2T_1$

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>write(A)</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>
Serializable schedules

- Serializable schedules are the ‘good schedules’
- Parallel executions of transactions
- But still maintain illusion of serial execution
- DBMS should ensure that only serializable schedules occur
- This is usually done using locks
Detecting non-serializability

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>write(A)</td>
<td>read(A)</td>
<td>write(C)</td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td>write(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>read(C)</td>
<td></td>
</tr>
</tbody>
</table>
• A cycle, so not conflict serializable

\[ T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \]

• **Theorem.** A schedule is conflict serializable if and only if its precedence graph is acyclic
Another example

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>write(C)</td>
<td>write(B)</td>
<td>read(C)</td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td>read(D)</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
<td>write(D)</td>
</tr>
</tbody>
</table>
• Equivalent serial schedules

\[ T_1T_2T_3T_4 \]
\[ T_1T_3T_2T_4 \]
Yet another example

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Not conflict serializable
- But result the same as running

$$T_2T_1T_3$$
View serializability

- Two schedules are **view equivalent** if
  - Corresponding reads in the two schedules always read the same value
  - The changes made to the database are always the same
- A schedule is **view serializable** if it is view equivalent to a serial schedule
- Schedule on previous slide is view serializable
View serializability

- The following is not view serializable

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(B)</td>
<td>read(B)</td>
<td>write(A)</td>
</tr>
<tr>
<td>read(A)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- To see this need to check all serial combinations
View serializability

- We have two notions of serializability
- Conflict serializable schedules are also view serializable
  - (because swapping commuting operations does not change behaviour of schedule)
- View serializable schedules need not be conflict serializable
  - (see example a few slides back)
Summary

• ACID requirements for databases
  - Atomicity, consistency, isolation, durability

• Isolation is an illusion
  - In reality transactions are evaluated in parallel

• Two notions of good schedules
  - Conflict serializability
  - View serializability

• For exam you should be able to use these