Announcements

- HW6 due on Today
- WQ7 (last!) due on Sunday
- HW7 will be posted tomorrow
  - due on Wed, May 24
  - using JDBC to execute SQL from Java
  - using SQL Server via Azure
  - setup covered in section tomorrow

Outline

- Serial and Serializable Schedules (18.1)
- Conflict Serializability (18.2)
- Locks (18.3)

Review: Transactions

- Problem: An application must perform several writes and reads to the database, as a unit
- Solution: multiple actions of the application are bundled into one unit called a Transaction

Review: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (i.e., looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database
Isolation: The Problem

• Multiple transactions are running concurrently \( T_1, T_2, \ldots \)
• They read/write some common elements \( A_1, A_2, \ldots \)
• How can we prevent unwanted interference?
• The SCHEDULER is responsible for that

Notation says nothing about tables... (These techniques apply more generally.)

Serial Schedule

• A serial schedule is one in which transactions are executed one after the other, in some sequential order
• Fact: nothing can go wrong if the system executes transactions serially
  – But database systems don’t do that because we need better performance

A schedule is a sequence of interleaved actions from all transactions

Example

\[
\begin{align*}
T1 & \\
\text{READ}(A, t) & \quad \text{READ}(A, s) \\
\quad t := t+100 & \quad s := s^2 \\
\text{WRITE}(A, t) & \quad \text{WRITE}(A, s) \\
\text{READ}(B, t) & \quad \text{READ}(B, s) \\
\quad t := t+100 & \quad s := s^2 \\
\text{WRITE}(B, t) & \quad \text{WRITE}(B, s)
\end{align*}
\]

A and B are elements in the database \( t \) and \( s \) are variables in txn source code

A Serial Schedule

\[
\begin{array}{c|c}
T1 & T2 \\
\text{READ}(A, t) & \text{READ}(A, s) \\
\quad t := t+100 & \quad s := s^2 \\
\text{WRITE}(A, t) & \text{WRITE}(A, s) \\
\text{READ}(B, t) & \text{READ}(B, s) \\
\quad t := t+100 & \quad s := s^2 \\
\text{WRITE}(B, t) & \text{WRITE}(B, s)
\end{array}
\]

Another Serial Schedule

\[
\begin{array}{c|c}
T1 & T2 \\
\text{READ}(A, t) & \text{READ}(A, s) \\
\quad t := t+100 & \quad s := s^2 \\
\text{WRITE}(A, t) & \text{WRITE}(A, s) \\
\text{READ}(B, t) & \text{READ}(B, s) \\
\quad t := t+100 & \quad s := s^2 \\
\text{WRITE}(B, t) & \text{WRITE}(B, s)
\end{array}
\]

Time
A schedule is **serializable** if it is equivalent to some serial schedule.

A Non-Serializable Schedule:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td></td>
</tr>
<tr>
<td>s := s'2</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td>WRITE(B, t)</td>
</tr>
<tr>
<td>WRITE(B, s)</td>
<td></td>
</tr>
</tbody>
</table>

This is a serializable schedule.

This is NOT a serial schedule.

How do We Know if a Schedule is Serializable?

**Notation**

- T<sub>i</sub>: r<sub>i</sub>(A); w<sub>i</sub>(A); r<sub>i</sub>(B); w<sub>i</sub>(B)
- T<sub>j</sub>: r<sub>j</sub>(A); w<sub>j</sub>(A); r<sub>j</sub>(B); w<sub>j</sub>(B)

**Key Idea:** Focus on conflicting operations.

Conflicts:

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW

Conflict Serializability:

- **Conflicts:** (it means: cannot be swapped)
  - Two actions by same transaction T<sub>i</sub>: r<sub>i</sub>(X); w<sub>i</sub>(Y)
  - Two writes by T<sub>i</sub>, T<sub>j</sub> to same element: w<sub>i</sub>(X); w<sub>j</sub>(X)
  - Read/write by T<sub>i</sub>, T<sub>j</sub> to same element: r<sub>i</sub>(X); w<sub>j</sub>(X)
Conflict Serializability

• A schedule is conflict serializable if it can be transformed into a serial schedule by a series of swaps of adjacent non-conflicting actions

• Every conflict-serializable schedule is serializable

• A serializable schedule may not necessarily be conflict-serializable

Example:

\[
\begin{align*}
\text{r}_1(\text{A}); & \text{ w}_1(\text{A}); \text{ r}_2(\text{A}); \text{ w}_2(\text{A}); \text{ r}_1(\text{B}); \text{ w}_1(\text{B}); \text{ r}_2(\text{B}); \\
\text{r}_1(\text{A}); & \text{ w}_1(\text{A}); \text{ r}_1(\text{B}); \text{ w}_1(\text{B}); \text{ r}_2(\text{A}); \text{ w}_2(\text{A}); \text{ r}_2(\text{B}); \text{ w}_2(\text{B})
\end{align*}
\]

Testing for Conflict-Serializability

Precedence graph:

• A node for each transaction \( T_i \),

• An edge from \( T_i \) to \( T_j \) whenever an action in \( T_i \) conflicts with, and comes before an action in \( T_j \)

• The schedule is serializable iff the precedence graph is acyclic
Example 1

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_3(A); r_2(B); w_2(B) \]

\[ 1 \quad 2 \quad 3 \]

This schedule is \textit{conflict-serializable}

Example 2

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_3(A); r_2(B); w_2(B) \]

\[ 1 \quad B \quad 2 \quad A \quad 3 \]

This schedule is \textit{NOT conflict-serializable}

Scheduler

- \textbf{Scheduler} = is the module that schedules the transaction’s actions, ensuring serializability
- Also called \textbf{Concurrency Control Manager}
- We discuss next how a scheduler may be implemented

Implementing a Scheduler

Major differences between database vendors
- \textbf{Locking Scheduler}
  - Aka “pessimistic concurrency control”
  - SQLite, SQL Server, DB2, Spanner
- \textbf{Multiversion Concurrency Control (MVCC)}
  - Aka “optimistic concurrency control”
  - Postgres, Oracle, Spanner

We discuss only locking in 414
Locking Scheduler

Simple idea:
• Each element has a unique lock
• Each transaction must first acquire the lock before reading/writing that element
• If the lock is taken by another transaction, then wait
• The transaction must release the lock(s)

By using locks scheduler ensures conflict-serializability

What Data Elements are Locked?

Major differences between vendors:
• Lock on the entire database
  – SQLite
• Lock on individual records
  – SQL Server, DB2, etc

Let's Study SQLite First

• SQLite is very simple
• More info: http://www.sqlite.org/atomiccommit.html

• Lock types
  – READ LOCK (to read)
  – RESERVED LOCK (to write)
  – PENDING LOCK (wants to commit)
  – EXCLUSIVE LOCK (to commit)

SQLite

Step 1: when a transaction begins
• Acquire a READ LOCK (aka "SHARED" lock)
• All these transactions may read happily
• They all read data from the database file
• If the transaction commits without writing anything, then it simply releases the lock

SQLite

Step 2: when one transaction wants to write
• Acquire a RESERVED LOCK
• May coexists with many READ LOCKs
• Writer TXN may write; these updates are only in main memory; others don't see the updates
• Reader TXN continue to read from the file
• New readers accepted
• No other TXN is allowed a RESERVED LOCK

SQLite

Step 3: when writer transaction wants to commit, it needs exclusive lock, which can't coexists with read locks
• Acquire a PENDING LOCK
• May coexists with old READ LOCKs
• No new READ LOCKS are accepted
• Wait for all read locks to be released
SQLite

Step 4: when all read locks have been released
- Acquire the EXCLUSIVE LOCK
- Nobody can touch the database now
- All updates are written permanently to the database file
- Release the lock and COMMIT

Demonstrating Locking in SQLite

T1:
begin transaction;
select * from R;
-- T1 has a READ LOCK

T2:
begin transaction;
select * from R;
-- T2 asked for a RESERVED LOCK: DENIED

T3:
begin transaction;
select * from R;
commit;
-- everything works fine, could obtain READ LOCK
Demonstrating Locking in SQLite

T1:
  commit;
  -- SQL error: database is locked
  -- T1 asked for PENDING LOCK -- GRANTED
  -- T1 asked for EXCLUSIVE LOCK -- DENIED

T3':
  begin transaction;
  select * from R;
  -- T3 asked for READ LOCK-- DENIED (due to T1)

T2:
  commit;
  -- releases the last READ LOCK; T1 can commit