TRANSACTION MANAGEMENT
[CH 16]
Transaction Management

Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);

You

Bank Balance : $100

Read Balance: $100

Sufficient funds?
Pay $25
New balance: $75

Bank Balance : $75!
Transaction Management

- Inconsistency
  - Interleaving actions of different user programs
  - System crash/user abort/…

- Provide the users an illusion of a single-user system
  - Could insist on admitting only one query into the system at any time
    - lower utilization: CPU/IO overlap
    - long running queries starve other queries

Read \((A)\);
Check \((A > $25)\);
Pay \($25\);
\(A = A - 25\);
Write \((A)\);
What is a Transaction?

• Collection of operations that form a single logical unit
  – A sequence of many actions considered to be one atomic unit of work

• Logical unit:
  – begin transaction .... (SQL) end transaction

• Operations:
  – Read (X), Write (X): Assume R/W on tuples (can be relaxed)
  – Special actions: begin, commit, abort

• Desirable Property: Must leave the DB in a consistent state
  – (DB is consistent when the transaction begins)
  – Consistency: DBMS only enforces ICs specified by the user
  – DBMS does not understand any other semantics of the data
The ACID Properties

- **Atomicity:** All actions in the Xact happen, or none happen.
- **Consistency:** Consistent DB + consistent Xact ⇒ consistent DB
- **Isolation:** Execution of one Xact is isolated from that of other Xacts.
- **Durability:** If a Xact commits, its effects persist.

```
Begin
  Read (A);
  A = A - 25;
  Write (A);
  Read (B);
  B = B + 25;
  Write (B);
Commit
```
### Schedules

- **Schedule**: An interleaving of actions from a set of Xacts, where the actions of any one Xact are in the original order.
  - Actions of Xacts *as seen by the DB*
  - *Complete schedule*: each Xact ends in commit or abort
  - *Serial schedule*: No interleaving of actions from different Xacts.
- **Initial State + Schedule → Final State**

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>begin</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>R(C)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(C)</td>
<td>commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>abort</td>
</tr>
</tbody>
</table>

**Example:**

\[
\begin{align*}
\text{T1} & \quad R(A) \quad W(A) \\
\text{T2} & \quad \text{commit} \\
\text{begin} & \quad R(B) \quad W(B) \\
\text{R(C)} & \quad W(C) \\
\end{align*}
\]

**Time**

**Notation:**

- **T**: Transaction
- **R**: Read
- **W**: Write
- **commit**: Commit
- **abort**: Abort
Acceptable Schedules

• One sensible “isolated, consistent” schedule:
  – Run Xacts one at a time (serial schedule)

• **Serializable** schedules:
  – Final state is what *some complete* serial schedule of *committed* transactions would have produced.
  – Can different serial schedules have different final states?
    • Yes, all are “OK”!
  – Aborted Xacts?
    • ignore them for a little while (made to ‘disappear’ using logging)
  – Other external actions (besides R/W to DB)
    • e.g. print a computed value, fire a missile, ...
    • Assume (for this class) these values are written to the DB, and can be undone
Serializability Violations

- @Start \((A, B) = (1000, 100)\)
  - End \((990, 210)\)
- T1→T2:
  - \((900, 200) \rightarrow (990, 220)\)
- T2→T1:
  - \((1100, 110) \rightarrow (1000, 210)\)
  - **W-R conflict:** Dirty read
    - *Could* lead to a non-serializable execution
  - Also R-W and W-W conflicts

<table>
<thead>
<tr>
<th>T1: Transfer $100 from A to B</th>
<th>T2: Add 10% interest to A &amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>R(A) /A -= 100</td>
<td>R(A) /A *= 1.1</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>R(B) /B *= 1.1</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
<tr>
<td>R(B) /B += 100</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More Conflicts

• **RW Conflicts (Unrepeatable Read)**
  – $R_{T2}(X) \rightarrow W_{T1}(X)$, T1 overwrites what T2 read.
  – $R_{T2}(X) \rightarrow W_{T1}(X) \rightarrow R_{T2}(X)$. T2 sees a different X value!

• **WW Conflicts (Overwriting Uncommitted Data)**
  – T2 overwrites what T1 wrote.
    • E.g. : Students in the same group get the same project grade.
    • $T_p: W(X=A), W(Y=A)$  \hspace{1cm} $T_{TA}: W(X=B), W(Y=B)$
    • $W_p(X=A) \rightarrow W_{TA}(X=B) \rightarrow W_{TA}(Y=B) \rightarrow W_p(Y=A)$
      [Note: no reads]
  – Usually occurs in conjunction with other anomalies.
    • Unless you have “blind writes”.


Now, Aborted Transactions

• **Serializable schedule**: Equivalent to a serial schedule of *committed* Xacts.
  – as if aborted Xacts *never happened*.

• Two Issues:
  – How does one undo the effects of a Xact?
    • We’ll cover this in logging/recovery
  – What if another Xact sees these effects??
    • Must undo that Xact as well!
Cascading Aborts

- Abort of T1 requires abort of T2!
  - Cascading Abort

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>commit</td>
</tr>
<tr>
<td>abort</td>
<td></td>
</tr>
</tbody>
</table>
Cascading Aborts

- Abort of T1 requires abort of T2!
  - Cascading Abort

- Consider commit of T2
  - Can we undo T2?

- Recoverable schedule: Commit only after all xacts that supply dirty data have committed.
Cascading Aborts

- **ACA (avoids cascading abort) schedule**
  - Transaction only reads committed data
  - One in which cascading abort cannot arise.
  - Schedule is also recoverable

```
<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>commit</td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>
```
Locking: A Technique for C. C.

- Concurrency control usually done via locking.
- Lock info maintained by a “lock manager”:
  - Stores (XID, RID, Mode) triples.
    - This is a simplistic view; suffices for now.
  - Mode ∈ {S,X}
    - Lock compatibility table:

```
<table>
<thead>
<tr>
<th></th>
<th>--</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>S</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>
```

- If a Xact can’t get a lock
  - Suspended on a wait queue
- When are locks acquired?
  - Buffer manager call!
Two-Phase Locking (2PL)

- **2PL:**
  - If T wants to read (modify) an object, first obtains an S (X) lock
  - If T releases any lock, it can acquire no new locks!
  - Guarantees serializability! Why?

- **Strict 2PL:**
  - Hold all locks until end of Xact
  - Guarantees serializability, and ACA too!
    - Note ACA schedules are always recoverable
Schedule with Locks

<table>
<thead>
<tr>
<th>T1: Transfer $100 from A to B</th>
<th>T2: Add 10% interest to A &amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td>R(A) /A -= 100</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(A) /A *= 1.1</td>
<td>R(B)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(B)</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
<tr>
<td>R(B) /B += 100</td>
<td>U_x(A), U_x(B)/commit</td>
</tr>
<tr>
<td>W(B)</td>
<td>R(A)</td>
</tr>
<tr>
<td>commit</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>…</td>
</tr>
</tbody>
</table>
Deadlocks

- Deadlocks can cause the system to wait forever.
- Need to detect deadlock and break, or prevent deadlocks
- Simple mechanism: timeout and abort
- More sophisticated methods exist

\[ X_{T_1}(B), X_{T_2}(A), S_{T_1}(A), S_{T_2}(B) \]
Precedence Graph

• Precedence (or Serializability) graph:
  – Nodes = Committed Xacts
  – Conflicts = Arcs

■ Conflict equivalent:
  ■ Same sets of actions
  ■ Conflicting actions in the same order

■ Conflict serializable: Conflict equivalent to a serial schedule
Schedule with Locks

<table>
<thead>
<tr>
<th>T1: Transfer $100 from A to B</th>
<th>T2: Add 10% interest to A &amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td>R(A) /A -= 100</td>
<td>X(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>R(A) /A *= 1.1</td>
<td>W(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>R(B) /B *= 1.1</td>
</tr>
<tr>
<td>commit</td>
<td>W(B)</td>
</tr>
<tr>
<td>R(B) /B += 100</td>
<td>commit</td>
</tr>
<tr>
<td>W(B)</td>
<td>U_x(A), U_x(B)/commit</td>
</tr>
</tbody>
</table>

T1: Transfer $100 from A to B
begin
R(A) /A -= 100
W(A)
R(B) /B *= 1.1
W(B)
commit
R(B) /B += 100
W(B)
commit

T2: Add 10% interest to A & B
begin
X(A)
R(A)
W(A)
X(A) – Wait!
R(A)
W(A)
...
Conflict Serializability & Graphs

Theorem: A schedule is conflict serializable iff its precedence graph is acyclic

Theorem: 2PL ensures that the precedence graph will be acyclic

• Why Strict 2PL?
  – Guarantees ACA
    • read only committed values
  – How? Write locks until EOT
    • No WW or WR => on abort replace original value
Deadlocks

$X_{T_1}(B), X_{T_2}(A), S_{T_1}(A), S_{T_2}(B)$

- Deadlocks can cause the system to wait forever.
- Need to detect deadlock and break, or prevent deadlocks
- Detect deadlock
  - Draw a lock graph. Cycles implies a deadlock
- Alternative ways of dealing with deadlock
  - Break Deadlock
    - On each lock request “update the lock graph”. If a cycle is detected, abort one of the transactions. The aborted transaction is restarted after waiting for a time-out interval.
  - Prevent deadlock
    - Assign priorities to the transactions. If a transaction, T1, requests a lock that is being held by another transaction, T2, with a lower priority, then T1 “snatches” the lock from T2 by aborting T2 (which frees up the lock on the resource). T2 is then restarted again after a time-out.
Transaction Support in SQL

• Transaction boundary
  – Begin implicitly, or end by *Commit work, Rollback work*
  – For long running transactions: *Savepoint*

• Transaction characteristics
  – Diagnostic size: # error messages...
  – Access mode: Read only, Read Write
  – Isolation level
    • Serializable: default
    • Repeatable reads:
      – Read only committed records
      – Between two reads by the same Xact, no updates by another Xact
    • Read committed
      – Read only committed records
    • Read uncommitted
      – Read only committed records

(not in the official course syllabus)
Phantom Problem

• T1: Scan Sailors for the oldest sailor for ratings 1 and 2
  – Assume that at the start the oldest sailor with rating 1 has age 80, oldest sailor with rating 2 has age 90, and the second oldest sailor with rating 2 is 85 years old
  – T1 identifies pages with sailors having a rating 1, and locks these pages. It computes the first tuple (rating = 1, oldest-age = 80)
  – T1 then gets ready to lock pages with sailor tuples with rating 2. However, before it can get started, T2 arrives

• T2: Inserts a tuple with rating 1 and age 99, and deletes the oldest sailor with rating 2 (whose age is 90)
  – The new tuple is inserted into a page that doesn’t have a sailor with rating 1 or 2, and is not locked by T1
  – T2 commits

• T1 now resumes and completes looking at sailors with rating 2.
• The final answer produced by T1 is (1,80) (2,85) does not correspond to either of the two serial schedules:
  – T1 -> T2 Answer: (1, 80), (2, 90)
  – T2 -> T1 Answer: (1, 99), (2, 85)
Transaction and Constraints

Create Table A (akey, bref, ...)
Create Table B (bkey, aref, ...)

Q: How to insert the first tuple, either in A or B?

• Solution:
  – Insert tuples in the same transaction
  – Defer the constraint checking

• SQL constraint modes
  – DEFERRED: Check at commit time.
  – IMMEDIATE: Check immediately
The **ACID** Properties

- **Atomicity**: All actions in the Xact happen, or none happen.

- **Consistency**: Consistent DB + consistent Xact $\Rightarrow$ consistent DB

- **Isolation**: Execution of one Xact is isolated from that of other Xacts.

- **Durability**: If a Xact commits, its effects persist.