Transactions; Concurrency; Recovery

Amol Deshpande
CMSC424

Databases

- Data Models
  - Conceptual representation of the data
- Data Retrieval
  - How to ask questions of the database
  - How to answer those questions
- Data Storage
  - How/where to store data, how to access it
- Data Integrity
  - Manage crashes, concurrency
  - Manage semantic inconsistencies
Overview

- **Transaction**: A sequence of database actions enclosed within special tags
- Properties:
  - **Atomicity**: Entire transaction or nothing
  - **Consistency**: Transaction, executed completely, takes database from one consistent state to another
  - **Isolation**: Concurrent transactions appear to run in isolation
  - **Durability**: Effects of committed transactions are not lost
- Consistency: Transaction programmer needs to guarantee that
  - DBMS can do a few things, e.g., enforce constraints on the data
- Rest: DBMS guarantees

How does..

- .. this relate to *queries* that we discussed?
  - Queries don’t update data, so *durability* and *consistency* not relevant
  - Would want *concurrency*
    - Consider a query computing total balance at the end of the day
  - Would want *isolation*
    - What if somebody makes a *transfer* while we are computing the balance
    - Typically not guaranteed for such long-running queries

- TPC-C vs TPC-H
### Assumptions and Goals

**Assumptions:**
- The system can crash at any time
- Similarly, the power can go out at any point
  - Contents of the main memory won’t survive a crash, or power outage
- BUT… *disks are durable. They might stop, but data is not lost.*
  - For now.
- Disks only guarantee *atomic sector writes*, nothing more
- Transactions are by themselves consistent

**Goals:**
- Guaranteed durability, atomicity
- As much concurrency as possible, while not compromising isolation and/or consistency
  - Two transactions updating the same account balance… NO
  - Two transactions updating different account balances… YES

---

### Next…

- States of a transaction
- A simple solution called *shadow copy*
  - Satisfies Atomicity, Durability, and Consistency, but no Concurrency
  - Very inefficient
Transaction states

- Make updates on a copy of the database.
- Switch pointers atomically after done.
  - Some text editors work this way

Shadow Copy

- Make updates on a copy of the database.
- Switch pointers atomically after done.
  - Some text editors work this way
Shadow Copy

- Atomicity:
  - As long as the DB pointer switch is atomic.
    - Okay if DB pointer is in a single block
- Concurrency:
  - No.
- Isolation:
  - No concurrency, so isolation is guaranteed.
- Durability:
  - Assuming disk is durable (we will assume this for now).
- Very inefficient:
  - Databases tend to be very large. Making extra copies not feasible.
    - Further, no concurrency.

Next...

- Concurrency control schemes
  - A CC scheme is used to guarantee that concurrency does not lead to problems
  - For now, we will assume durability is not a problem
    - So no crashes
    - Though transactions may still abort
- Schedules
- When is concurrency okay?
  - Serial schedules
  - Serializability
A Schedule

Transactions:
T1: transfers $50 from A to B
T2: transfers 10% of A to B
Database constraint: A + B is constant (checking+saving accts)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A = A - 50</td>
<td>tmp = A \times 0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A - tmp</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
</tr>
<tr>
<td>B = B + 50</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>B = B + tmp</td>
</tr>
</tbody>
</table>

Effect: Before After
A 100 45
B 50 105

Each transaction obeys the constraint.
This schedule does too.

Schedules

- A **schedule** is simply a (possibly interleaved) execution sequence of transaction instructions

- **Serial Schedule**: A schedule in which transaction appear one after the other
  - ie., No interleaving

- Serial schedules satisfy isolation and consistency
  - Since each transaction by itself does not introduce inconsistency
**Example Schedule**

Another “serial” schedule:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Effect:</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A = A - 50</td>
<td></td>
<td>A</td>
<td>100</td>
<td>40</td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td>B</td>
<td>50</td>
<td>110</td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = B + 50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Consistent?
Constraint is satisfied.

Since each Xion is consistent, any serial schedule must be consistent.

**Another schedule**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Is this schedule okay?</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A = A - 50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = B + 50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

read(A)
A = A * 0.1
A = A - tmp
write(A)
read(B)
B = B + tmp
write(B)

Is this schedule okay?

Lets look at the final effect...

Effect: | Before | After |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
<td>45</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
<td>105</td>
</tr>
</tbody>
</table>

Consistent.
So this schedule is okay too.
### Another schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Is this schedule okay?</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>A = A - 50</td>
<td>tmp = A * 0.1</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A - tmp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>B = B + 50</td>
<td>B = B + tmp</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
<td></td>
</tr>
</tbody>
</table>

#### Effect:

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
</tr>
</tbody>
</table>

Further, the effect same as the serial schedule 1. Called **serializable**

### Example Schedules (Cont.)

#### A “bad” schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Effect:</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
<td>A</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>A = A - 50</td>
<td>tmp = A * 0.1</td>
<td>B</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A - tmp</td>
<td></td>
<td>60</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td>read(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td>B = B + tmp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = B + 50</td>
<td>write(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B = B + tmp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Not consistent
Serializability

- A schedule is called *serializable* if its final effect is the same as that of a *serial schedule*

- Serializability → schedule is fine and does not result in inconsistent database
  - Since serial schedules are fine

- Non-serializable schedules are unlikely to result in consistent databases

- We will ensure serializability
  - Typically relaxed in real high-throughput environments

Serializability

- Not possible to look at all $n!$ serial schedules to check if the effect is the same
  - Instead we ensure serializability by allowing or not allowing certain schedules

- Conflict serializability

- View serializability
  - View serializability allows more schedules
## Conflict Serializability

- Two read/write instructions “conflict” if
  - They are by different transactions
  - They operate on the same data item
  - At least one is a “write” instruction

- Why do we care?
  - If two read/write instructions don’t conflict, they can be “swapped” without any change in the final effect
  - However, if they conflict they CAN’T be swapped without changing the final effect

### Equivalence by Swapping

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>A = A -50</td>
<td>read(A)</td>
<td>A = A -50</td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td>B = B+50</td>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td>write(B)</td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td>B = B+ tmp</td>
<td>read(B)</td>
<td>B = B+ tmp</td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td>write(B)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Effect</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
<td>45</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
<td>105</td>
</tr>
</tbody>
</table>

==

<table>
<thead>
<tr>
<th>Effect</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
<td>45</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
<td>105</td>
</tr>
</tbody>
</table>
Equivalence by Swapping

\[
\begin{array}{c|c|c|c|c|c}
T1 & T2 & T1 & T2 \\
\hline
\text{read(A)} & \text{read(A)} & \text{read(A)} & \text{read(A)} \\
A = A -50 & \text{tmp} = A \times 0.1 & A = A - \text{tmp} & \text{tmp} = A \times 0.1 \\
\text{write(A)} & \text{write(A)} & A = A - \text{tmp} & A = A - \text{tmp} \\
\hline
\text{read(B)} & \text{read(B)} & \text{read(B)} & \text{read(B)} \\
B = B +50 & \text{write(A)} & B = B +50 & \text{write(A)} \\
\text{write(B)} & \text{write(B)} & B = B + \text{tmp} & B = B + \text{tmp} \\
\end{array}
\]

Effect:
<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
</tr>
</tbody>
</table>

Effect:
<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
</tr>
</tbody>
</table>

Conflict Serializability

- Conflict-equivalent schedules:
  - If S can be transformed into S’ through a series of swaps, S and S’ are called conflict-equivalent
  - conflict-equivalent guarantees same final effect on the database

- A schedule S is conflict-serializable if it is conflict-equivalent to a serial schedule
**Equivalence by Swapping**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>A = A -50</td>
<td></td>
<td>A = A -50</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tmp = A*0.1</td>
<td></td>
<td>tmp = A*0.1</td>
</tr>
<tr>
<td></td>
<td>A = A – tmp</td>
<td></td>
<td>A = A – tmp</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B=B+50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td>write(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect:</td>
<td>Before</td>
<td>After</td>
<td>Effect:</td>
</tr>
<tr>
<td>A</td>
<td>100</td>
<td>45</td>
<td>=</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
<td>105</td>
<td>=</td>
</tr>
</tbody>
</table>

**Equivalence by Swapping**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>A = A -50</td>
<td></td>
<td>A = A -50</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tmp = A*0.1</td>
<td></td>
<td>tmp = A*0.1</td>
</tr>
<tr>
<td></td>
<td>A = A – tmp</td>
<td></td>
<td>A = A – tmp</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B=B+50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td>write(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect:</td>
<td>Before</td>
<td>After</td>
<td>Effect:</td>
</tr>
<tr>
<td>A</td>
<td>100</td>
<td>45</td>
<td>=</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
<td>105</td>
<td>=</td>
</tr>
</tbody>
</table>
A “bad” schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A) A = A -50</td>
<td>read(A) tmp = A*0.1 A = A – tmp write(A) read(B)</td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>read(B) B=B+50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
</tbody>
</table>

X

Y

Can’t move Y below X
read(B) and write(B) conflict

Other options don’t work either

So: Not Conflict Serializable

---

**Serializable**

- In essence, following set of instructions is not conflict-serializable:

<table>
<thead>
<tr>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(Q)</td>
<td></td>
</tr>
<tr>
<td>write(Q)</td>
<td>write(Q)</td>
</tr>
</tbody>
</table>
**View-Serializability**

- Similarly, following not conflict-serializable

<table>
<thead>
<tr>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($Q$)</td>
<td>write($Q$)</td>
<td>write($Q$)</td>
</tr>
<tr>
<td>write($Q$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- BUT, it is serializable
  - Intuitively, this is because the **conflicting write instructions don’t matter**
  - The final write is the only one that matters

- View-serializability allows these
  - Read up

**Other notions of serializability**

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td></td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td></td>
</tr>
<tr>
<td>write($A$)</td>
<td></td>
</tr>
<tr>
<td>read($B$)</td>
<td></td>
</tr>
<tr>
<td>$B := B - 10$</td>
<td></td>
</tr>
<tr>
<td>write($B$)</td>
<td></td>
</tr>
</tbody>
</table>

- Not conflict-serializable or view-serializable, but serializable
- Mainly because of the +/- only operations
  - Requires analysis of the actual operations, not just read/write operations
- Most high-performance transaction systems will allow these
Testing for conflict-serializability

- Given a schedule, determine if it is conflict-serializable

- Draw a precedence-graph over the transactions
  - A directed edge from T1 and T2, if they have conflicting instructions, and T1’s conflicting instruction comes first

- If there is a cycle in the graph, not conflict-serializable
  - Can be checked in at most $O(n + e)$ time, where $n$ is the number of vertices, and $e$ is the number of edges

- If there is none, conflict-serializable

- Testing for view-serializability is NP-hard.

Example Schedule (Schedule A) + Precedence Graph

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read(Y)</td>
<td>read(X)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(Z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(U)</td>
<td></td>
<td>write(Y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>write(Z)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(U)</td>
<td></td>
<td></td>
<td>read(V)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(U)</td>
<td></td>
<td></td>
<td>read(W)</td>
<td>write(W)</td>
</tr>
</tbody>
</table>

[Diagram of precedence graph with transactions $T_1$ to $T_5$.]
Recap so far...

- We discussed:
  - Serial schedules, serializability
  - Conflict-serializability, view-serializability
  - How to check for conflict-serializability

- We haven’t discussed:
  - How to guarantee serializability?
    - Allowing transactions to run, and then aborting them if the schedules wasn’t serializable is clearly not the way to go
  - We instead use schemes to guarantee that the schedule will be conflict-serializable
  - Also, recoverability?

---

### Recoverability

- Serializability is good for consistency

- But what if transactions fail?
  - T2 has already committed
    - A user might have been notified
  - Now T1 abort creates a problem
    - T2 has seen its effect, so just aborting T1 is not enough. T2 must be aborted as well (and possibly restarted)
    - But T2 is committed

---

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A = A -50</td>
<td>tmp = A*0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A – tmp</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>B=B+50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
<tr>
<td>ABORT</td>
<td></td>
</tr>
</tbody>
</table>
Recoverability

- Recoverable schedule: If T1 has read something T2 has written, T2 must commit before T1
  - Otherwise, if T1 commits, and T2 aborts, we have a problem

- Cascading rollbacks: If T10 aborts, T11 must abort, and hence T12 must abort and so on.

<table>
<thead>
<tr>
<th>$T_{10}$</th>
<th>$T_{11}$</th>
<th>$T_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>write($A$)</td>
<td>write($A$)</td>
</tr>
</tbody>
</table>

Dirty read: Reading a value written by a transaction that hasn’t committed yet

- Cascadeless schedules:
  - A transaction only reads committed values.
  - So if T1 has written A, but not committed it, T2 can’t read it.
    - No dirty reads

- Cascadeless $\rightarrow$ No cascading rollbacks
  - That’s good
  - We will try to guarantee that as well
Recap so far...

- We discussed:
  - Serial schedules, serializability
  - Conflict-serializability, view-serializability
  - How to check for conflict-serializability
  - Recoverability, cascade-less schedules

- We haven’t discussed:
  - How to guarantee serializability?
    - Allowing transactions to run, and then aborting them if the schedules wasn’t serializable is clearly not the way to go
  - We instead use schemes to guarantee that the schedule will be conflict-serializable

Concurrency Control

Amol Deshpande
CMSC424
Approach, Assumptions etc..

- **Approach**
  - Guarantee conflict-serializability by allowing certain types of concurrency
    - Lock-based

- **Assumptions:**
  - Durability is not a problem
    - So no crashes
    - Though transactions may still abort

- **Goal:**
  - Serializability
  - Minimize the bad effect of aborts (cascade-less schedules only)

Lock-based Protocols

- A transaction *must* get a lock before operating on the data

- Two types of locks:
  - *Shared* (S) locks (also called *read locks*)
    - Obtained if we want to only read an item
  - *Exclusive* (X) locks (also called *write locks*)
    - Obtained for updating a data item
Lock instructions

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock

Example schedule:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td>B ← B-50</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>display(A+B)</td>
</tr>
<tr>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>A ← A + 50</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
</tbody>
</table>

Lock instructions

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock

Example schedule:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(B)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td>B ← B-50</td>
<td>unlock(A)</td>
</tr>
<tr>
<td>write(B)</td>
<td>lock-S(B)</td>
</tr>
<tr>
<td>unlock(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>lock-X(A)</td>
<td>unlock(B)</td>
</tr>
<tr>
<td>read(A)</td>
<td>display(A+B)</td>
</tr>
<tr>
<td>A ← A + 50</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>unlock(A)</td>
<td></td>
</tr>
</tbody>
</table>
Lock-based Protocols

- Lock requests are made to the concurrency control manager
  - It decides whether to grant a lock request
- T1 asks for a lock on data item A, and T2 currently has a lock on it?
  - Depends

<table>
<thead>
<tr>
<th>T2 lock type</th>
<th>T1 lock type</th>
<th>Should allow?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>Shared</td>
<td>YES</td>
</tr>
<tr>
<td>Shared</td>
<td>Exclusive</td>
<td>NO</td>
</tr>
<tr>
<td>Exclusive</td>
<td>-</td>
<td>NO</td>
</tr>
</tbody>
</table>

- If compatible, grant the lock, otherwise T1 waits in a queue.

Lock-based Protocols

- How do we actually use this to guarantee serializability/recoverability?
  - Not enough just to take locks when you need to read/write something

T1
lock-X(B)
read(B)
B ← B-50
write(B)
unlock(B)
lock-X(A)
read(A)
A ← A + 50
write(A)
unlock(A)

lock-X(A), lock-X(B)
A = A-50
B = B+50
unlock(A), unlock(B)
2-Phase Locking Protocol (2PL)

- Phase 1: Growing phase
  - Transaction may obtain locks
  - But may not release them

- Phase 2: Shrinking phase
  - Transaction may only release locks

- Can be shown that this achieves conflict-serializability
  - `lock-point`: the time at which a transaction acquired last lock
  - if `lock-point(T1) < lock-point(T2)`, there can’t be an edge from T2 to T1 in the precedence graph

```
lock-X(B)  
read(B)    
B ← B - 50
write(B)   
lock-X(A)  
read(A)    
A ← A + 50
write(A)   
unlock(A) 
```

2 Phase Locking

- Example: T1 in 2PL

```
<table>
<thead>
<tr>
<th>T1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lock-X(B)</td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td></td>
<td>B ← B - 50</td>
</tr>
<tr>
<td></td>
<td>write(B)</td>
</tr>
<tr>
<td></td>
<td>lock-X(A)</td>
</tr>
<tr>
<td></td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>A ← A + 50</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td></td>
<td>unlock(B)</td>
</tr>
<tr>
<td></td>
<td>unlock(A)</td>
</tr>
</tbody>
</table>
```

Growing phase

Shrinking phase
2 Phase Locking

- Guarantees *conflict-serializability*, but not cascade-less recoverability

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A), lock-S(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(A)</td>
<td>lock-X(A)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>unlock(A), unlock(B)</td>
<td>unlock(A)</td>
<td>unlock(A)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td>Commit</td>
</tr>
<tr>
<td>&lt;action fails&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2 Phase Locking

- Guarantees *conflict-serializability*, but not cascade-less recoverability

- Guaranteeing just recoverability:
  - ★ If T2 reads a dirty data of T1 (ie, T1 has not committed), then T2 can’t commit unless T1 either commits or aborts
  - ★ If T1 commits, T2 can proceed with committing
  - ★ If T1 aborts, T2 must abort
    - ➢ So cascades still happen
Strict 2PL

- Release exclusive locks only at the very end, just before commit or abort

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A), lock-S(B)</td>
<td>lock-X(A)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(A)</td>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td>unlock(A)</td>
<td>unlock(A)</td>
</tr>
<tr>
<td>unlock(A), unlock(B)</td>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>

<ction fails>

Works. Guarantees cascade-less and recoverable schedules.

Strict 2PL

- Release exclusive locks only at the very end, just before commit or abort
  - Read locks are not important

- Rigorous 2PL: Release both exclusive and read locks only at the very end
  - The serializability order === the commit order
  - More intuitive behavior for the users
    - No difference for the system
Strict 2PL

- Lock conversion:
  - Transaction might not be sure what it needs a write lock on
  - Start with a S lock
  - *Upgrade* to an X lock later if needed
  - Doesn't change any of the other properties of the protocol

Implementation of Locking

- A separate process, or a separate module
- Uses a *lock table* to keep track of currently assigned locks and the requests for locks
  - Read up in the book
Recap so far...

- Concurrency Control Scheme
  - A way to guarantee serializability, recoverability etc

- Lock-based protocols
  - Use locks to prevent multiple transactions accessing the same data items

- 2 Phase Locking
  - Locks acquired during growing phase, released during shrinking phase

- Strict 2PL, Rigorous 2PL

More Locking Issues: Deadlocks

- No xction proceeds:
  Deadlock
  - T1 waits for T2 to unlock A
  - T2 waits for T1 to unlock B

  Rollback transactions
  Can be costly...

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lock-X(A)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>B ← B-50</td>
<td>lock-S(B)</td>
</tr>
<tr>
<td></td>
<td>write(B)</td>
<td></td>
</tr>
</tbody>
</table>
2PL and Deadlocks

- 2PL does not prevent deadlock
  - Strict doesn’t either

- > 2 xctions involved?
  - Rollbacks expensive

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(B)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td>B ← B-50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td>lock-S(B)</td>
</tr>
<tr>
<td>lock-X(A)</td>
<td></td>
</tr>
</tbody>
</table>

Preventing deadlocks

- Solution 1: A transaction must acquire all locks before it begins
  - Not acceptable in most cases

- Solution 2: A transaction must acquire locks in a particular order over the data items
  - Also called graph-based protocols

- Solution 3: Use time-stamps; say T1 is older than T2
  - wait-die scheme: T1 will wait for T2. T2 will not wait for T1; instead it will abort and restart
  - wound-wait scheme: T1 will wound T2 (force it to abort) if it needs a lock that T2 currently has; T2 will wait for T1.

- Solution 4: Timeout based
  - Transaction waits a certain time for a lock; aborts if it doesn’t get it by then
Deadlock detection and recovery

- Instead of trying to prevent deadlocks, let them happen and deal with them if they happen.
- How do you detect a deadlock?
  - Wait-for graph
  - Directed edge from $T_i$ to $T_j$
    - $T_i$ waiting for $T_j$

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S(V)$</td>
<td>$X(V)$</td>
<td>$X(Z)$</td>
<td>$X(W)$</td>
<td></td>
</tr>
<tr>
<td>$S(W)$</td>
<td>$S(W)$</td>
<td>$S(V)$</td>
<td>$S(V)$</td>
<td></td>
</tr>
</tbody>
</table>

Suppose $T_4$ requests lock-$S(Z)$.

Dealing with Deadlocks

- Deadlock detected, now what?
  - Will need to abort some transaction
  - Prefer to abort the one with the minimum work done so far
  - Possibility of starvation
    - If a transaction is aborted too many times, it may be given priority in continueing
Locking granularity

- Locking granularity
  - What are we taking locks on? Tables, tuples, attributes?

- Coarse granularity
  - e.g. take locks on tables
  - less overhead (the number of tables is not that high)
  - very low concurrency

- Fine granularity
  - e.g. take locks on tuples
  - much higher overhead
  - much higher concurrency
  - What if I want to lock 90% of the tuples of a table?
    - Prefer to lock the whole table in that case

The highest level in the example hierarchy is the entire database.
The levels below are of type area, file or relation and record in that order.
Can lock at any level in the hierarchy
Granularity Hierarchy

- New lock mode, called *intentional* locks
  - Declare an intention to lock parts of the subtree below a node
  - IS: *intention shared*
    - The lower levels below may be locked in the shared mode
  - IX: *intention exclusive*
  - SIX: *shared and intention-exclusive*
    - The entire subtree is locked in the shared mode, but I might also want to get exclusive locks on the nodes below

- Protocol:
  - If you want to acquire a lock on a data item, all the ancestors must be locked as well, at least in the intentional mode
  - So you always start at the top root node

---

(1) Want to lock $F_a$ in shared mode, $DB$ and $A1$ must be locked in at least IS mode (but IX, SIX, S, X are okay too)

(2) Want to lock rc1 in exclusive mode, $DB$, $A2$, $Fc$ must be locked in at least IX mode (SIX, X are okay too)
Granularity Hierarchy

<table>
<thead>
<tr>
<th>Parent locked in</th>
<th>Child can be locked in</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>IS, S</td>
</tr>
<tr>
<td>IX</td>
<td>IS, S, IX, X, SIX</td>
</tr>
<tr>
<td>S</td>
<td>[S, IS] not necessary</td>
</tr>
<tr>
<td>SIX</td>
<td>X, IX, [SIX]</td>
</tr>
<tr>
<td>X</td>
<td>none</td>
</tr>
</tbody>
</table>

Compatibility Matrix with Intention Lock Modes

The compatibility matrix (which locks can be present simultaneously on the same data item) for all lock modes is:

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>S IX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>holder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S IX</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>X</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Can T2 access object f2.2 in X mode?
What locks will T2 get?
Examples

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then repeatedly gets an S lock on tuples of R, and occasionally upgrades to X on the tuples.
- T2 uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can use lock escalation to decide which.

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recap, Next…

- Deadlocks
  - Detection, prevention, recovery
- Locking granularity
  - Arranged in a hierarchy
  - Intentional locks
- Next…
  - Brief discussion of some other concurrency schemes
Other CC Schemes

- Time-stamp based
  - Transactions are issued time-stamps when they enter the system
  - The time-stamps determine the *serializability* order
  - So if T1 entered before T2, then T1 should be before T2 in the serializability order
  - Say $timestamp(T1) < timestamp(T2)$
  - If T1 wants to read data item A
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is *aborted*
  - If T1 wants to write data item A
    - If a transaction with larger time-stamp already read that data item or written it, then the write is *rejected* and T1 is aborted
  - Aborted transaction are restarted with a new timestamp
    - Possibility of *starvation*

---

Other CC Schemes

- Time-stamp based
  - Example

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(Y)</td>
<td>read(Y)</td>
<td>write(Y)</td>
<td>write(Z)</td>
<td>read(X)</td>
<td></td>
</tr>
<tr>
<td>read(X)</td>
<td>abort</td>
<td>write(Z)</td>
<td>abort</td>
<td>write(Y)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>write(Z)</td>
<td></td>
</tr>
</tbody>
</table>
Other CC Schemes

- Time-stamp based
  - As discussed here, has too many problems
    - Starvation
    - Non-recoverable
    - Cascading rollbacks required
  - Most can be solved fairly easily
    - Read up
  - Remember: We can always put more and more restrictions on what the transactions can do to ensure these things
    - The goal is to find the minimal set of restrictions to as to not hinder concurrency

Other CC Schemes

- Optimistic concurrency control
  - Also called validation-based

  - Intuition
    - Let the transactions execute as they wish
    - At the very end when they are about to commit, check if there might be any problems/conflicts etc
      - If no, let it commit
      - If yes, abort and restart

  - Optimistic: The hope is that there won’t be too many problems/aborts

- Rarely used any more
The “Phantom” problem

- An interesting problem that comes up for dynamic databases
- Schema: accounts(branchname, acct_no, balance, …)
- Transaction 1: Find the maximum balance in each branch
  - Both maximum entries in the corresponding branches
- Execution sequence:
  - T1 locks all tuples corresponding to “branch1”, finds the maximum balance and releases the locks
  - T2 does its two insert/deletes
  - T1 locks all tuples corresponding to “branch2”, finds the maximum balance and releases the locks
- Not serializable

Recovery

Amol Deshpande
CMSC424
**Context**

- ACID properties:
  - We have talked about Isolation and Consistency
  - How do we guarantee Atomicity and Durability?
    - Atomicity: Two problems
      - Part of the transaction is done, but we want to cancel it
        - ABORT/ROLLBACK
      - System crashes during the transaction. Some changes made it to the disk, some didn’t.
    - Durability:

- Essentially similar solutions

**Reasons for crashes**

- Transaction failures
  - Logical errors, deadlocks
- System crash
  - Power failures, operating system bugs etc
- Disk failure
  - Head crashes; *for now we will assume*
    - **STABLE STORAGE:** Data never lost. Can approximate by using RAID and maintaining geographically distant copies of the data
Approach, Assumptions etc.

- Approach:
  - Guarantee A and D:
    - by controlling how the disk and memory interact,
    - by storing enough information during normal processing to recover from failures
    - by developing algorithms to recover the database state
  - Assumptions:
    - System may crash, but the disk is durable
    - The only atomicity guarantee is that a disk block write is atomic
  - Once again, obvious naïve solutions exist that work, but that are too expensive.
    - E.g. The shadow copy solution we saw earlier
      - Make a copy of the database; do the changes on the copy; do an atomic switch of the dbpointer at commit time
    - Goal is to do this as efficiently as possible

STEAL vs NO STEAL, FORCE vs NO FORCE

- STEAL:
  - The buffer manager can steal a (memory) page from the database
    - i.e., it can write an arbitrary page to the disk and use that page for something else from the disk
    - In other words, the database system doesn’t control the buffer replacement policy
  - Why a problem?
    - The page might contain dirty writes, i.e., writes/updates by a transaction that hasn’t committed
  - But, we must allow steal for performance reasons.

- NO STEAL:
  - Not allowed. More control, but less flexibility for the buffer manager.
### STEAL vs NO STEAL, FORCE vs NO FORCE

**FORCE:**
- The database system *forces* all the updates of a transaction to disk before committing
- **Why?**
  - To make its updates permanent before committing
- **Why a problem?**
  - Most probably random I/Os, so poor response time and throughput
  - Interferes with the disk controlling policies

**NO FORCE:**
- Don’t do the above. Desired.
- **Problem:**
  - Guaranteeing durability becomes hard
  - We might still have to *force* some pages to disk, but minimal.

### STEAL vs NO STEAL, FORCE vs NO FORCE: Recovery implications

<table>
<thead>
<tr>
<th>Force</th>
<th>No Force</th>
<th>Desired</th>
<th>Trivial</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Steal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Steal</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
STEAL vs NO STEAL, FORCE vs NO FORCE: Recovery implications

How to implement A and D when No Steal and Force?

- Only updates from committed transaction are written to disk (since no steal)
- Updates from a transaction are forced to disk before commit (since force)
  - A minor problem: how do you guarantee that all updates from a transaction make it to the disk atomically?
  - Remember we are only guaranteed an atomic block write
  - What if some updates make it to disk, and other don’t?
- Can use something like shadow copying/shadow paging

- No atomicity/durability problem arise.

Terminology

Deferred Database Modification:
- Similar to NO STEAL, NO FORCE
  - Not identical
- Only need redos, no undos
- We won’t cover this today

Immediate Database Modification:
- Similar to STEAL, NO FORCE
- Need both redos, and undos
Log-based Recovery

- Most commonly used recovery method
- Intuitively, a log is a record of everything the database system does
- For every operation done by the database, a log record is generated and stored typically on a different (log) disk
- \(<T1, \text{START}>\)
- \(<T2, \text{COMMIT}>\)
- \(<T2, \text{ABORT}>\)
- \(<T1, A, 100, 200>\)
  - T1 modified A; old value = 100, new value = 200

Log

- Example transactions \(T_0\) and \(T_1\) (\(T_0\) executes before \(T_1\)):
  - \(T_0\): read (A)
    - A: - A - 50
    - write (A)
    - read (B)
    - B: - B + 50
    - write (B)
  - \(T_1\): read (C)
    - C: - C - 100
    - write (C)

- Log:
  
  \(<T_0 \text{ start}>\)  \(<T_0 \text{ start}>\)  \(<T_0 \text{ start}>\)  
  \(<T_0, A, 950>\)  \(<T_0, A, 950>\)  \(<T_0, A, 950>\)  
  \(<T_0, B, 2050>\)  \(<T_0, B, 2050>\)  \(<T_0, B, 2050>\)  
  \(<T_0 \text{ commit}>\)  \(<T_0 \text{ commit}>\)  \(<T_0 \text{ commit}>\)  
  \(\text{T_1 start}\)  \(\text{T_1 start}\)  \(\text{T_1 start}\)  
  \(\text{T_1, C, 600}\)  \(\text{T_1, C, 600}\)  \(\text{T_1, C, 600}\)  

(a)  (b)  (c)
Log-based Recovery

- Assumptions:
  1. Log records are immediately pushed to the disk as soon as they are generated
  2. Log records are written to disk in the order generated
  3. A log record is generated before the actual data value is updated
  4. Strict two-phase locking
     - The first assumption can be relaxed
     - As a special case, a transaction is considered committed only after the $<T_1, \text{COMMIT}>$ has been pushed to the disk

- But, this seems like exactly what we are trying to avoid??
  - Log writes are sequential
  - They are also typically on a different disk

- Aside: LFS == log-structured file system

Log-based Recovery

- Assumptions:
  1. Log records are immediately pushed to the disk as soon as they are generated
  2. Log records are written to disk in the order generated
  3. A log record is generated before the actual data value is updated
  4. Strict two-phase locking
     - The first assumption can be relaxed
     - As a special case, a transaction is considered committed only after the $<T_1, \text{COMMIT}>$ has been pushed to the disk

- NOTE: As a result of assumptions 1 and 2, if data item A is updated, the log record corresponding to the update is always forced to the disk before data item A is written to the disk
  - This is actually the only property we need; assumption 1 can be relaxed to just guarantee this (called write-ahead logging)
**Using the log to ***abort/rollback***

- **STEAL** is allowed, so changes of a transaction may have made it to the disk

- **UNDO**(T1):
  - Procedure executed to *rollback/undo* the effects of a transaction
  - E.g.
    - `<T1, START>`
    - `<T1, A, 200, 300>`
    - `<T1, B, 400, 300>`
    - `<T1, A, 300, 200>`  
      [note: second update of A]
    - T1 decides to abort

  - Any of the changes might have made it to the disk

---

**Using the log to ***abort/rollback***

- **UNDO**(T1):
  - Go *backwards* in the log looking for log records belonging to T1
  - Restore the values to the old values
  - NOTE: Going backwards is important.
    - A was updated twice
  - In the example, we simply:
    - Restore A to 300
    - Restore B to 400
    - Restore A to 200

  - Note: No other transaction better have changed A or B in the meantime
    - *Strict two-phase locking*
We don’t require FORCE, so a change made by the committed transaction may not have made it to the disk before the system crashed

- BUT, the log record did (recall our assumptions)

REDO(T1):
- Procedure executed to recover a committed transaction
- E.g.
  - `<T1, START>`
  - `<T1, A, 200, 300>`
  - `<T1, B, 400, 300>`
  - `<T1, A, 300, 200>  [[ note: second update of A ]]`
  - `<T1, COMMIT>`

  By our assumptions, all the log records made it to the disk (since the transaction committed)
  But any or none of the changes to A or B might have made it to disk

---

REDO(T1):
- Go *forwards* in the log looking for log records belonging to T1
- Set the values to the new values
- NOTE: Going forwards is important.
- In the example, we simply:
  - Set A to 300
  - Set B to 300
  - Set A to 200
Idempotency

- Both redo and undo are required to idempotent
  - $F$ is idempotent, if $F(x) = F(F(x)) = F(F(F(F(\ldots F(x))))))$
- Multiple applications shouldn’t change the effect
  - This is important because we don’t know exactly what made it to the disk, and we can’t keep track of that
  - E.g. consider a log record of the type
    - $<$T1, A, incremented by 100$>$
    - Old value was 200, and so new value was 300
  - But the on disk value might be 200 or 300 (since we have no control over the buffer manager)
  - So we have no idea whether to apply this log record or not
  - Hence, value based logging is used (also called physical), not operation based (also called logical)

Log-based recovery

- Log is maintained
- If during the normal processing, a transaction needs to abort
  - UNDO() is used for that purpose
- If the system crashes, then we need to do recovery using both UNDO() and REDO()
  - Some transactions that were going on at the time of crash may not have completed, and must be aborted/undone
  - Some transaction may have committed, but their changes didn’t make it to disk, so they must be redone
  - Called restart recovery
Restart Recovery (after a crash)

- After restart, go backwards into the log, and make two lists
  - How far ?? For now, assume till the beginning of the log.

- undo_list: A list of transactions that must be undone
  - \(<T_i, \text{START}>\) record is in the log, but no \(<T_i, \text{COMMIT}>\)

- redo_list: A list of transactions that need to be redone
  - Both \(<T_i, \text{START}>\) and \(<T_i, \text{COMMIT}>\) records are in the log

- After that:
  - UNDO all the transactions on the undo_list one by one
  - REDO all the transaction on the redo_list one by one

Restart Recovery (after a crash)

- Must do the UNDOs first before REDO
  - \(<T_1, A, 10, 20>\)
  - \(<T_1, \text{abort}>\) \([\text{so A was restored back to 10}]\)
  - \(<T_2, A, 10, 30>\)
  - \(<T_2, \text{commit}>\)

- If we do UNDO(T1) first, and then REDO(T2), it will be okay
- Trying to do other way around doesn’t work

- NOTE: In reality, most system generate special log records when transactions are aborted, and in that case, they have to do REDO before UNDO
  - However, our scheme doesn’t, so we must do UNDO before REDO
Checkpointing

How far should we go back in the log while constructing redo and undo lists??

- It is possible that a transaction made an update at the very beginning of the system, and that update never made it to disk
  - very very unlikely, but possible (because we don’t do force)
- For correctness, we have to go back all the way to the beginning of the log
- Bad idea !!

Checkpointing is a mechanism to reduce this

Checkpointing

Periodically, the database system writes out everything in the memory to disk

- Goal is to get the database in a state that we know (not necessarily consistent state)

Steps:

- Stop all other activity in the database system
- Write out the entire contents of the memory to the disk
  - Only need to write updated pages, so not so bad
  - Entire === all updates, whether committed or not
- Write out all the log records to the disk
- Write out a special log record to disk
  - <CHECKPOINT LIST_OF_ACTIVE_TRANSACTIONS>
  - The second component is the list of all active transactions in the system right now
- Continue with the transactions again
## Restart Recovery w/ checkpoints

- **Key difference:** Only need to go back till the last checkpoint
- **Steps:**
  - **undo_list:**
    - Go back till the checkpoint as before.
    - Add all the transactions that were active at that time, and that didn’t commit
      - e.g. possible that a transactions started before the checkpoint, but didn’t finish till the crash
  - **redo_list:**
    - Similarly, go back till the checkpoint constructing the redo_list
    - Add all the transactions that were active at that time, and that did commit
  - Do UNDOs and REDOs as before

## Recap so far …

- **Log-based recovery**
  - Uses a *log* to aid during recovery
- **UNDO()**
  - Used for normal transaction abort/rollback, as well as during restart recovery
- **REDO()**
  - Used during restart recovery
- **Checkpoints**
  - Used to reduce the restart recovery time
Write-ahead logging

- We assumed that log records are written to disk as soon as generated
  - Too restrictive
- Write-ahead logging:
  - Before an update on a data item (say A) makes it to disk, the log records referring to the update must be forced to disk
  - How?
    - Each log record has a log sequence number (LSN)
      - Monotonically increasing
    - For each page in the memory, we maintain the LSN of the last log record that updated a record on this page
      - pageLSN
    - If a page \( P \) is to be written to disk, all the log records till \( pageLSN(P) \) are forced to disk

Write-ahead logging

- Write-ahead logging (WAL) is sufficient for all our purposes
  - All the algorithms discussed before work

- Note the special case:
  - A transaction is not considered committed, unless the \( <T, \text{commit}> \) record is on disk
Other issues

- The system halts during checkpointing
  - Not acceptable
  - Advanced recovery techniques allow the system to continue processing while checkpointing is going on

- System may crash during recovery
  - Our simple protocol is actually fine
  - In general, this can be painful to handle

- B+-Tree and other indexing techniques
  - Strict 2PL is typically not followed (we didn’t cover this)
  - So physical logging is not sufficient; must have logical logging

 Other issues

- ARIES: Considered the canonical description of log-based recovery
  - Used in most systems
  - Has many other types of log records that simplify recovery significantly

- Loss of disk:
  - Can use a scheme similar to checkpointing to periodically dump the database onto tapes or optical storage
  - Techniques exist for doing this while the transactions are executing (called fuzzy dumps)

- Shadow paging:
  - Read up
We studied how to do STEAL and NO FORCE through log-based recovery scheme.

ACID Properties
- Atomicity and Durability:
  - Logs, undo(), redo(), WAL etc

- Consistency and Isolation:
  - Concurrency schemes

- Strong interactions:
  - We had to assume Strict 2PL for proving correctness of recovery