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
A transaction is a unit of program execution that accesses and possibly updates various data items.

E.g. transaction to transfer $50 from account A to account B:

1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

Two main issues to deal with:

- Failures of various kinds, such as hardware failures and system crashes
- Concurrent execution of multiple transactions
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
States of a transaction
**Transaction State**

- **Active** – the initial state; the transaction stays in this state while it is executing.
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - ★ restart the transaction
    - ➢ can be done only if no internal logical error
  - ★ kill the transaction
- **Committed** – after successful completion.
Transaction states

- Partially committed
- Committed
- Active
- Failed
- Aborted
Concurrency: Why?
- Increased processor and disk utilization
- Reduced average response times

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>tmp = A*0.1</td>
</tr>
<tr>
<td>A = A - 50</td>
<td>A = A – tmp</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>B = B + 50</td>
<td>B = B + tmp</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</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 transactions appear one after the other.
  - i.e., 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>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A = A -50</td>
<td>tmp = A*0.1</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>A = A - tmp</td>
<td>100</td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td>tmp = A*0.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>B = B+ tmp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>After</td>
</tr>
<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>
<tr>
<td></td>
<td>Effect:</td>
<td>Before</td>
</tr>
<tr>
<td></td>
<td>Consistent ?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Constraint is satisfied.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Since each Xion is consistent, any serial schedule must be consistent</td>
<td></td>
</tr>
</tbody>
</table>
Another schedule

<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>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>B = B + 50</td>
<td>B = B + tmp</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
</tr>
</tbody>
</table>

Is this schedule okay?

Lets look at the final effect...

Effect:

<table>
<thead>
<tr>
<th></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>

Consistent.

So this schedule is okay too.
Another schedule

<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></td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>B = B + 50</td>
<td>B = B + tmp</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
</tr>
</tbody>
</table>

Is this schedule okay?

Let's look at the final effect...

Effect:

<table>
<thead>
<tr>
<th></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>

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>50</td>
</tr>
<tr>
<td>A = A -50</td>
<td>tmp = A*0.1</td>
<td>B</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>A = A – tmp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B = B+ tmp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(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 $\rightarrow$ 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

T1
---
read(A)
A = A -50
write(A)

T2
---
read(A)
tmp = A*0.1
A = A – tmp
write(A)

read(B)
B = B+50
write(B)

read(B)
B = B+ tmp
write(B)

Effect:
---
Before: A 100  B 50
After:   A 45   B 105

Effect:
---
Before: A 100  B 50
After:   A 45   B 105
## Equivalence by Swapping

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

### Effect:

<table>
<thead>
<tr>
<th></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>

Effect: $A \neq B$
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>read(A)</td>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>$A = A - 50$</td>
<td>$A = A - 50$</td>
<td>$A = A - 50$</td>
<td>$A = A - 50$</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>read(B)</td>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>$B = B + 50$</td>
<td>$B = B + 50$</td>
<td>$B = B + 50$</td>
<td>$B = B + 50$</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
<td>write(B)</td>
<td>write(B)</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>

==

<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>
## Equivalence by Swapping

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

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

<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></td>
<td>A = A – tmp</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>read(B)</td>
</tr>
<tr>
<td>read(B)</td>
<td>B = B+ tmp</td>
</tr>
<tr>
<td>B = B+50</td>
<td>write(B)</td>
</tr>
</tbody>
</table>

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

Other options don’t work either

So: Not Conflict 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>write($Q$)</td>
</tr>
<tr>
<td>write($Q$)</td>
<td></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></td>
</tr>
<tr>
<td>write($Q$)</td>
<td></td>
<td>write($Q$)</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></th>
<th>$T_1$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read($A$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td></td>
<td>$A := A - 50$</td>
<td>$B := B - 10$</td>
</tr>
<tr>
<td></td>
<td>write($A$)</td>
<td>write($B$)</td>
</tr>
<tr>
<td></td>
<td>read($B$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B := B + 50$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write($B$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>read($A$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$A := A + 10$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write($A$)</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>read(Y)</td>
<td>read(X)</td>
<td>read(Y)</td>
<td>write(Z)</td>
<td>read(Y)</td>
<td>read(V)</td>
</tr>
<tr>
<td>read(Z)</td>
<td></td>
<td>write(Y)</td>
<td></td>
<td>write(Y)</td>
<td>read(W)</td>
</tr>
<tr>
<td>read(U)</td>
<td></td>
<td></td>
<td>write(Z)</td>
<td></td>
<td>read(Z)</td>
</tr>
<tr>
<td>read(U)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>write(Z)</td>
</tr>
<tr>
<td>write(U)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Precedence Graph

- $T_1 \rightarrow T_2$
- $T_2 \rightarrow T_3$
- $T_3 \rightarrow T_4$
- $T_4 \rightarrow T_1$
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>write(A)</td>
</tr>
<tr>
<td></td>
<td>A = A – tmp</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
</tr>
<tr>
<td></td>
<td></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>ABORT</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>
<tr>
<td>write($A$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recoverability

- 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 → 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></th>
<th>T1</th>
<th></th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read(B)</td>
<td></td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>B ← B - 50</td>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td>write</td>
<td>write(B)</td>
<td></td>
<td>display(A+B)</td>
</tr>
<tr>
<td>read</td>
<td>read(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A ← A + 50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>write(A)</td>
<td></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></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(B)</td>
<td></td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td>read(A)</td>
</tr>
<tr>
<td>B ← B-50</td>
<td></td>
<td>unlock(A)</td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td>lock-S(B)</td>
</tr>
<tr>
<td>unlock(B)</td>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td>lock-X(A)</td>
<td></td>
<td>unlock(B)</td>
</tr>
<tr>
<td>read(A)</td>
<td></td>
<td>display(A+B)</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>unlock(A)</td>
<td></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), lock-X(B)
A = A - 50
B = B + 50
unlock(A), unlock(B)

lock-X(A)
read(A)
A ← A + 50
write(A)
unlock(A)
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 \( \text{lock-point}(T1) < \text{lock-point}(T2) \), there can’t be an edge from \( T2 \) to \( T1 \) in the precedence graph

T1
lock-X(B)
read(B)
write(B)
unlock(B)

lock-X(A)
read(A)
write(A)
unlock(A)
2 Phase Locking

- Example: T1 in 2PL

<table>
<thead>
<tr>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(B)</td>
</tr>
<tr>
<td>read(B)</td>
</tr>
<tr>
<td>$B \leftarrow B - 50$</td>
</tr>
<tr>
<td>write(B)</td>
</tr>
<tr>
<td>lock-X(A)</td>
</tr>
<tr>
<td>read(A)</td>
</tr>
<tr>
<td>$A \leftarrow A - 50$</td>
</tr>
<tr>
<td>write(A)</td>
</tr>
<tr>
<td>unlock(B)</td>
</tr>
<tr>
<td>unlock(A)</td>
</tr>
</tbody>
</table>

Growing phase

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

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

*<xction 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
Lock Table

- Black rectangles indicate granted locks, white ones indicate waiting requests.
- Lock table also records the type of lock granted or requested.
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks.
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted.
- If transaction aborts, all waiting or granted requests of the transaction are deleted.

atherine lock manager may keep a list of locks held by each transaction, to implement this efficiently.
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 transaction 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>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>lock-S(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>lock-X(A)</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 Ti to Tj
  - Ti waiting for Tj

Suppose T4 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 continuing
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>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S IX</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
Example

R1

t1

\[ T_1(\text{IS}) \]

\[ T_1(\text{S}) \]

\[ T_2(\text{IX}) \]

\[ T_2(\text{X}) \]

t2

t3

t4
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.

```
  --   IS   IX   S   X
  --   ✓    ✓    ✓    ✓    ✓
  IS    ✓    ✓    ✓    ✓
  IX    ✓    ✓    ✓
  S     ✓    ✓    ✓
  X     ✓
```
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></td>
<td>read($Y$)</td>
<td>read($Y$)</td>
<td>write($Y$)</td>
<td>read($X$)</td>
<td>read($X$)</td>
</tr>
<tr>
<td></td>
<td>read($X$)</td>
<td>abort</td>
<td>write($Z$)</td>
<td>abort</td>
<td>WRITE $Z$</td>
</tr>
<tr>
<td></td>
<td>abort</td>
<td></td>
<td>abort</td>
<td></td>
<td>WRITE $Y$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WRITE $Z$</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 Schemes:
Optimistic Concurrency Control

- 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
Each transaction $T_i$ has 3 timestamps

- $\text{Start}(T_i)$: the time when $T_i$ started its execution
- $\text{Validation}(T_i)$: the time when $T_i$ entered its validation phase
- $\text{Finish}(T_i)$: the time when $T_i$ finished its write phase

Serializability order is determined by timestamp given at validation time, to increase concurrency.

- Thus $\text{TS}(T_i)$ is given the value of $\text{Validation}(T_i)$.

This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.

- because the serializability order is not pre-decided, and
- relatively few transactions will have to be rolled back.
Other Schemes: Optimistic Concurrency Control

If for all $T_i$ with $\text{TS}(T_i) < \text{TS}(T_j)$ either one of the following condition holds:

- $\text{finish}(T_i) < \text{start}(T_j)$
- $\text{start}(T_j) < \text{finish}(T_i) < \text{validation}(T_j)$ and the set of data items written by $T_i$ does not intersect with the set of data items read by $T_j$.

then validation succeeds and $T_j$ can be committed. Otherwise, validation fails and $T_j$ is aborted.

Justification: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and

- the writes of $T_j$ do not affect reads of $T_i$ since they occur after $T_i$ has finished its reads.
- the writes of $T_i$ do not affect reads of $T_j$ since $T_j$ does not read any item written by $T_i$. 
Other Schemes: Optimistic Concurrency Control

- Example of schedule produced using validation

<table>
<thead>
<tr>
<th>$T_{25}$</th>
<th>$T_{26}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ($B$)</td>
<td>read ($B$)</td>
</tr>
<tr>
<td></td>
<td>$B := B + 50$</td>
</tr>
<tr>
<td>read ($A$)</td>
<td>read ($A$)</td>
</tr>
<tr>
<td>$\langle validate \rangle$</td>
<td>$A := A + 50$</td>
</tr>
<tr>
<td>display ($A + B$)</td>
<td>$\langle validate \rangle$</td>
</tr>
<tr>
<td></td>
<td>write ($B$)</td>
</tr>
<tr>
<td></td>
<td>write ($A$)</td>
</tr>
</tbody>
</table>
Other CC Schemes: Snapshot Isolation

- Very popular scheme, used as the primary scheme by many systems including Oracle, PostgreSQL etc…
  - Several others support this in addition to locking-based protocol

- A type of “optimistic concurrency control”

- Key idea:
  - For each object, maintain past “versions” of the data along with timestamps
    - Every update to an object causes a new version to be generated
Other CC Schemes: Snapshot Isolation

Read queries:
- Let “t” be the “time-stamp” of the query, i.e., the time at which it entered the system
- When the query asks for a data item, provide a version of the data item that was latest as of “t”
  - Even if the data changed in between, provide an old version
- No locks needed, no waiting for any other transactions or queries
- The query executes on a consistent snapshot of the database

Update queries (transactions):
- Reads processed as above on a snapshot
- Writes are done in private storage
- At commit time, for each object that was written, check if some other transaction updated the data item since this transaction started
  - If yes, then abort and restart
  - If no, make all the writes public simultaneously (by making new versions)
A transaction $T_1$ executing with Snapshot Isolation

- takes snapshot of committed data at start
- always reads/modifies data in its own snapshot
- updates of concurrent transactions are not visible to $T_1$
- writes of $T_1$ complete when it commits

**First-committer-wins rule:**

- Commits only if no other concurrent transaction has already written data that $T_1$ intends to write.

### Table

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W(Y := 1)$ Commit</td>
<td>Start</td>
<td>$W(X := 2)$ Commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$R(X) \rightarrow 0$</td>
<td>$R(Y) \rightarrow 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$R(Z) \rightarrow 0$</td>
<td>$R(Y) \rightarrow 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$W(X := 3)$ Commit-Req</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$W(Z := 3)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Commit-Req</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abort</td>
<td></td>
</tr>
</tbody>
</table>

Concurrent updates not visible
Own updates are visible
Not first-committer of $X$
Serialization error, $T_2$ is rolled back
Other CC Schemes: Snapshot Isolation

- Advantages:
  - Read query don’t block at all, and run very fast
  - As long as conflicts are rare, update transactions don’t abort either
  - Overall better performance than locking-based protocols

- Major disadvantage:
  - Not serializable
  - Inconsistencies may be introduced
  - See the wikipedia article for more details and an example
Snapshot Isolation

- Example of problem with SI
  - T1: x := y
  - T2: y := x
  - Initially x = 3 and y = 17
    - Serial execution: x = ??, y = ??
    - if both transactions start at the same time, with snapshot isolation: x = ??, y = ??

- Called **skew write**

- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1
SI In Oracle and PostgreSQL

**Warning**: SI used when isolation level is set to serializable, by Oracle, and PostgreSQL versions prior to 9.1

- PostgreSQL’s implementation of SI (versions prior to 9.1) described in Section 26.4.1.3
- Oracle implements “first updater wins” rule (variant of “first committer wins”)
  - Concurrent writer check is done at time of write, not at commit time
  - Allows transactions to be rolled back earlier
  - Oracle and PostgreSQL < 9.1 do not support true serializable execution
- PostgreSQL 9.1 introduced new protocol called “Serializable Snapshot Isolation” (SSI)
  - Which guarantees true serializability including handling predicate reads (coming up)
The “Phantom” problem

- An interesting problem that comes up for dynamic databases
- Schema: accounts(acct_no, balance, zipcode, …)
- Transaction 1: Find the number of accounts in zipcode = 20742, and divide $1,000,000 between them
- Transaction 2: Insert <acctX, …, 20742, …>

Execution sequence:

- T1 locks all tuples corresponding to “zipcode = 20742”, finds the total number of accounts (= num_accounts)
- T2 does the insert
- T1 computes 1,000,000/num_accounts
- When T1 accesses the relation again to update the balances, it finds one new (“phantom”) tuples (the new tuple that T2 inserted)

- Not serializable
- See this for another example
Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
  - **Serializable**: is the default
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted**: allows even uncommitted data to be read

- In many database systems, read committed is the default consistency level
  - has to be explicitly changed to serializable when required
    - set isolation level serializable
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**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)

- **System crash**
  - Power failures, operating system bugs etc
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data

- **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
Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.

Block movements between disk and main memory are initiated through the following two operations:

- **input**(B) transfers the physical block B to main memory.
- **output**(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.

- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.
Example of Data Access

Buffer Block A
Buffer Block B

read(X)
write(Y)

input(A)
output(B)

work area of T₁
work area of T₂

memory
disk
Data Access (Cont.)

- Each transaction $T_i$ has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - $T_i$'s local copy of a data item $X$ is called $x_i$.
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read($X$)** assigns the value of data item $X$ to the local variable $x_i$.
  - **write($X$)** assigns the value of local variable $x_i$ to data item $\{X\}$ in the buffer block.
  - **Note:** output($B_X$) need not immediately follow write($X$). System can perform the output operation when it deems fit.

Transactions
- Must perform **read($X$)** before accessing $X$ for the first time (subsequent reads can be from local copy)
- **write($X$)** can be executed at any time before the transaction commits
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

No Force

Force

No Steal

Steal

Desired

Trivial
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 \textit{redos, no undos}
  ★ We won’t cover this today

■ Immediate Database Modification:
  ★ Similar to STEAL, NO FORCE
  ★ Need both \textit{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, START>`
- `<T2, COMMIT>`
- `<T2, ABORT>`
- `<T1, A, 100, 200>`
  - T1 modified A; old value = 100, new value = 200
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:

(a) $<T_0 \text{ start}>$

$<T_0, A, 950>$

$<T_0, B, 2050>$

(b) $<T_0 \text{ commit}>$

$<T_0 \text{ start}>$

(c) $<T_1 \text{ start}>$

$<T_1, C, 600>$

$<T_1 \text{ commit}>$
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 \(<T1, COMMIT>\) 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 <T1, 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.
    - \(<T1, \text{START}>\)
    - \(<T1, A, 200, 300>\)
    - \(<T1, B, 400, 300>\)
    - \(<T1, A, 300, 200>\)  
      
    - \(<T1, A, 300, 200>\)  
      
    - 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
  ★ Write a log record \( <T_i, X_j, V_i> \)
    ➢ such log records are called compensation log records
    ➢ \( <T1, A, 300>, <T1, B, 400>, <T1, A, 200> \)
  ★ Note: No other transaction better have changed A or B in the meantime
    ➢ Strict two-phase locking
Using the log to recover

- 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
Using the log to *recover*

- 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
  - $<T1, A, \text{incremented by 100}>$
  - 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
Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of all transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions

- **Redo phase**:
  1. Find last `<checkpoint L>` record, and set undo-list to L.
     - If no checkpoint record, start at the beginning
  2. Scan forward from above `<checkpoint L>` record
     1. Whenever a record `<T_i, X_j, V_1, V_2>` is found, redo it by writing `V_2` to `X_j`
     2. Whenever a log record `<T_i start>` is found, add `T_i` to undo-list
     3. Whenever a log record `<T_i commit>` or `<T_i abort>` is found, remove `T_i` from undo-list
Recovery Algorithm (Cont.)

**Undo phase:**

1. Scan log backwards from end
   1. Whenever a log record \(<T_i, X_j, V_1, V_2>\) is found where \(T_i\) is in undo-list perform same actions as for transaction rollback:
      1. perform undo by writing \(V_1\) to \(X_j\).
      2. write a log record \(<T_i, X_j, V_1>\)
   2. Whenever a log record \(<T_i \text{ start}>\) is found where \(T_i\) is in undo-list,
      1. Write a log record \(<T_i \text{ abort}>\)
      2. Remove \(T_i\) from undo-list
   3. Stop when undo-list is empty
      - i.e. \(<T_i \text{ start}>\) has been found for every transaction in undo-list

- After undo phase completes, normal transaction processing can commence
Example of Recovery

Begining of log
<T₀ start>
<T₀, B, 2000, 2050>
<T₁ start>
<checkpoint {T₀, T₁}>
<T₁, C, 700, 600>
<T₁ commit>
<T₂ start>
<T₂, A, 500, 400>
<T₀, B, 2000>
<T₀ abort>

End of log at crash!

Log records added during recovery

T₀ rollback (during normal operation) begins

T₀ rollback complete

T₂ is incomplete at crash

Undo Pass:
Undo list: T₂
T₂ rolled back in undo pass

Redo Pass:
Start log records found for all transactions in undo list
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 (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
Recap

- STEAL vs NO STEAL, FORCE vs NO FORCE
  - We studied how to do STEAL and NO FORCE through log-based recovery scheme
Recap

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