Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 - DB Applications

C. Faloutsos – A. Pavlo
Lecture#20: Overview of Transaction Management

Administrivia

• HW7 Phase 1: **Wed Nov 11th**
• HW7 Phase 2: **Mon Nov 30th**
• Recitations (always in WEH 5302):
  – **Tue Nov 10th @ 1:30pm**
  – **Tue Nov 17th @ 2:30pm**

Last Class

• Database Design
• Database Tuning

Today’s Class

• Transactions Overview
• Concurrency Control
• Recovery
Motivation

Lost Updates
Concurrency Control
• We both change the same record ("Smith"); how to avoid race condition?

Durability
Recovery
• You transfer $100 from savings→checking; power failure – what happens?

Concurrency Control & Recovery

• Valuable properties of DBMSs.
• Based on concept of transactions with ACID properties.

• Let’s talk about transactions…

Transactions

• A transaction is the execution of a sequence of one or more operations (e.g., SQL queries) on a shared database to perform some higher-level function.
• It is the basic unit of change in a DBMS:
  – Partial transactions are not allowed!

Transaction Example

• Move $100 from Andy’ bank account to his bookie’s account.
• Transaction:
  – Check whether Andy has $100.
  – Deduct $100 from his account.
  – Add $100 to his bookie’s account.
Strawman System

- Execute each txn one-by-one (i.e., serial order) as they arrive at the DBMS.
  - One and only one txn can be running at the same time in the DBMS.
- Before a txn starts, copy the entire database to a new file and make all changes to that file.
  - If the txn completes successfully, overwrite the original file with the new one.
  - If the txn fails, just remove the dirty copy.

Problem Statement

- Better approach is to allow concurrent execution of independent transactions.
- Q: Why do we want that?
  - Utilization/throughput (“hide” waiting for I/Os)
  - Increased response times to users.
- But we also would like:
  - Correctness
  - Fairness

Transactions

- Hard to ensure correctness…
  - What happens if Andy only has $100 and tries to pay off two bookies at the same time?
- Hard to execute quickly…
  - What happens if Andy needs to pay off his gambling debts very quickly all at once?

Problem Statement

- Arbitrary interleaving can lead to
  - Temporary inconsistency (ok, unavoidable)
  - Permanent inconsistency (bad!)
- Need formal correctness criteria.
Definitions

• A txn may carry out many operations on the data retrieved from the database
• However, the DBMS is only concerned about what data is read/written from/to the database.
  – Changes to the “outside world” are beyond the scope of the DBMS.

Formal Definitions

• **Database**: A fixed set of named data objects \((A, B, C, \ldots)\)
• **Transaction**: A sequence of read and write operations \((R(A), W(B), \ldots)\)
  – DBMS’s abstract view of a user program

Transactions in SQL

• A new txn starts with the `begin` command.
• The txn stops with either `commit` or `abort`:
  – If `commit`, all changes are saved.
  – If `abort`, all changes are undone so that it’s like as if the txn never executed at all.

Correctness Criteria: ACID

• **Atomicity**: All actions in the txn happen, or none happen.
• **Consistency**: If each txn is consistent and the DB starts consistent, then it ends up consistent.
• **Isolation**: Execution of one txn is isolated from that of other txns.
• **Durability**: If a txn commits, its effects persist.
Correctness Criteria: ACID

- **Atomicity**: “all or nothing”
- **Consistency**: “it looks correct to me”
- **Isolation**: “as if alone”
- **Durability**: “survive failures”

Overview

- Problem definition & ‘ACID’
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Atomicity of Transactions

- Two possible outcomes of executing a txn:
  - Txn might **commit** after completing all its actions.
  - or it could **abort** (or be aborted by the DBMS) after executing some actions.
- DBMS guarantees that txns are **atomic**.
  - From user’s point of view: txn always either executes all its actions, or executes no actions at all.

Mechanisms for Ensuring Atomicity

- We take $100 out of Andy’s account but then there is a power failure before we transfer it to his bookie.
- When the database comes back on-line, what should be the correct state of Andy’s account?
Mechanisms for Ensuring Atomicity

• One approach: **LOGGING**
  – DBMS logs all actions so that it can *undo* the actions of aborted transactions.
• Think of this like the black box in airplanes…

• Logging used by all modern systems.
  • Q: Why?
  • A: Audit Trail & Efficiency Reasons

• Another approach: **SHADOW PAGING**
  – DBMS makes copies of pages and txns make changes to those copies. Only when the txn commits is the page made visible to others.
  – Originally from System R.
• Few systems do this:
  – CouchDB
  – LMDB (OpenLDAP)
Overview

• Problem definition & ‘ACID’
• Atomicity
• Consistency
• Isolation
• Durability

Database Consistency

• Database Consistency: Data in the DBMS is accurate in modeling the real world and follows integrity constraints

Transaction Consistency

• Transaction Consistency: if the database is consistent before the txn starts (running alone), it will be after also.
• Transaction consistency is the application’s responsibility.
  – We won’t discuss this further...

Overview

• Problem definition & ‘ACID’
• Atomicity
• Consistency
• Isolation
• Durability
Isolation of Transactions

- Users submit txns, and each txn executes as if it was running by itself.
- Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- Q: How do we achieve this?

A: Many methods - two main categories:
- Pessimistic – Don’t let problems arise in the first place.
- Optimistic – Assume conflicts are rare, deal with them after they happen.

Example

- Consider two txns:
  - T1 transfers $100 from B’s account to A’s
  - T2 credits both accounts with 6% interest.

- Assume at first A and B each have $1000.
- Q: What are the legal outcomes of running T1 and T2?
Example

- **Q:** What are the possible outcomes of running T1 and T2 together?
- **A:** Many! But A+B should be: $2000 \times 1.06 = $2120
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect must be equivalent to these two transactions running **serially** in some order.

Legal outcomes:
- A=1166, B=954
- A=1160, B=960

The outcome depends on whether T1 executes before T2 or vice versa.

Serial Execution Example

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN A=A+100 B=B–100 COMMIT</td>
<td>BEGIN A=A<em>1.06 B=B</em>1.06 COMMIT</td>
</tr>
<tr>
<td>BEGIN A=A+100 B=B–100 COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

A=1166, B=954
A=1160, B=960

Interleaving Transactions

- We can also interleave the txns in order to maximize concurrency.
  - Slow disk/network I/O.
  - Multi-core CPUs.
### Interleaving Example (Good)

**Schedule**

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEGIN A=A+100</td>
<td>BEGIN B=B–100</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

**Effect**

A=1166, B=954

### Interleaving Example (Bad)

**Schedule**

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEGIN A=A+100</td>
<td>BEGIN B=B–100</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

**Effect**

A=1166, B=954

### Correctness

- **Q**: How do we judge that a schedule is correct?
- **A**: If it is equivalent to some serial execution

**Example**

The bank lost $6!
Formal Properties of Schedules

- **Serial Schedule**: A schedule that does not interleave the actions of different transactions.
- **Equivalent Schedules**: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.*

(* no matter what the arithmetic operations are!)

Serializability is a less intuitive notion of correctness compared to txn initiation time or commit order, but it provides the DBMS with significant additional flexibility in scheduling operations.

Serializable Schedule: A schedule that is equivalent to some serial execution of the transactions.

Note: If each transaction preserves consistency, every serializable schedule preserves consistency.

Interleaved Execution Anomalies

- **Read-Write** conflicts (R-W)
- **Write-Read** conflicts (W-R)
- **Write-Write** conflicts (W-W)

Q: Why not R-R conflicts?
Write-Read Conflicts

• Reading Uncommitted Data, “Dirty Reads”:

```
BEGIN R(A) W(A) R(B) W(B) ABORT
```

```
T1 T2

$10$12

$12$14

R(B) W(B)

ABORT
```

Read-Write Conflicts

• Unrepeatable Reads

```
BEGIN R(A) W(A) COMMIT
```

```
T1 T2

$10$

$10$19

R(A) W(A) COMMIT
```

Write-Write Conflicts

• Overwriting Uncommitted Data

```
BEGIN W(A) W(B) COMMIT
```

```
T1 T2

$10$

$19$

W(B) COMMIT

Bieber

Andy
```

Solution

• Q: How could you guarantee that all resulting schedules are correct (i.e., serializable)?

• A: Use locks!
Executing without Locks

- $T_1$: BEGIN R(A) W(A) COMMIT
- $T_2$: BEGIN R(A) W(A) COMMIT

Executing with Locks

- $T_1$: BEGIN LOCK(A) R(A) W(A) UNLOCK(A) COMMIT
- $T_2$: BEGIN LOCK(A) R(A) W(A) UNLOCK(A) COMMIT

Q: If a txn only needs to read ‘A’, should it still get a lock?
A: Yes, but you can get a shared lock.

Lock Types

- Basic Types:
  - $S$-LOCK – Shared Locks (reads)
  - $X$-LOCK – Exclusive Locks (writes)

Compatibility Matrix

<table>
<thead>
<tr>
<th></th>
<th>Shared</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>Exclusive</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
Executing with Locks

• Transactions request locks (or upgrades)
• Lock manager grants or blocks requests
• Transactions release locks
• Lock manager updates lock-table

• But this is not enough…

Concurreny Control

• We need to use a well-defined protocol that ensures that txns execute correctly.
• Two categories:
  – Two-Phase Locking (2PL)
  – Timestamp Ordering (T/O)

Overview

• Problem definition & ‘ACID’
• Atomicity
• Consistency
• Isolation
• Durability
Transaction Durability

- Records are stored on disk.
- For updates, they are copied into memory and flushed back to disk at the discretion of the O.S.
  - Unless forced-output: $W(B) \rightarrow \text{fsync()}$

This is slow! Nobody does this!

$\text{fsync()}$

- Kernel maintains a buffer cache between applications & disks.
  - If you just call $\text{write()}$, there is no guarantee that the data is durable on disk.
- Use $\text{fsync()}$ to force the OS to flush all modified in-core data to disk.
  - This blocks the thread until it completes.
  - Data may still live in on-disk cache but we cannot control that.

Buffer is added to output queue but is not flushed immediately
Write-Ahead Log

- Record the changes made to the database in a log before the change is made.
- Assume that the log is on stable storage.

Q: What to replicate?
- The complete page?
- Single tuple?

WAL Problems

- The log grows infinitely...
- We have to take checkpoints to reduce the amount of processing that we need to do.
- We will discuss this in further detail in upcoming classes.

The DBMS hasn’t flushed memory to disk at this point. We have to redo T1!

Safe to return result to application.
### ACID Properties

- **Atomicity**: All actions in the txn happen, or none happen.
- **Consistency**: If each txn is consistent, and the DB starts consistent, it ends up consistent.
- **Isolation**: Execution of one txn is isolated from that of other txns.
- **Durability**: If a txn commits, its effects persist.

### Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Concurrency control is automatic
  - System automatically inserts lock/unlock requests and schedules actions of different txns.
  - Ensures that resulting execution is equivalent to executing the txns one after the other in some order.

### Write-ahead logging (WAL) and the recovery protocol are used to:
- Undo the actions of aborted transactions.
- Restore the system to a consistent state after a crash.

### Overview

- **Atomicity**
- **Consistency**
- **Isolation**
- **Durability**
- **Concurrency Control**
- **Recovery**