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) is due Tues April 1st
• Recitations (always in SH 219):
  – Wed April 2nd 2:30-3:20
  – Wed April 9th 2:30-3:20

Last Class

• Database Design
• Database Tuning
Today’s Class
• Transactions Overview
• Concurrency Control
• Recovery

Motivation
• We both change the same record (“Smith”); how to avoid race condition?
• You transfer $100 from savings→checking; power failure – what happens?

Motivation
• We both change the same record (“Smith”); how to avoid race condition?
• You transfer $100 from savings→checking; power failure – what happens?

DBMSs automatically handle both issues: ‘transactions’
Concurrency Control & Recovery

- Valuable properties of DBMSs.
- Based on concept of transactions with **ACID** properties.
- Next lectures discuss these issues.

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 Christos’ bank account to his bookie’s account.*
- Transaction:
  - Check whether Christos 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.

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 Christos only has $100 and tries to pay off two bookies at the same time?*

- Hard to execute quickly…
  - *What happens if Christos 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.

A txn can abort itself or the DBMS can abort it.

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 Christos’ 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 Christos’ account?

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?
Mechanisms for Ensuring Atomicity

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

**What other mechanism can you think of?**

--

Mechanisms for Ensuring Atomicity

- 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.
- Nobody actually does this…

--

Overview

- Problem definition & ‘**ACID**’
- Atomicity
- **C**onsistency
- **I**solation
- **D**urability
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...*

Strong vs. Weak Consistency

- In a distributed DBMS, the consistency level determines when other nodes see new data in the database:
  - **Strong**: Guaranteed to see all writes immediately, but txns are slower.
  - **Weak/Eventual**: Will see writes at some later point in time, but txns are faster.
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?

Isolation of Transactions

• 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

\[
\begin{align*}
T1 & : & \text{BEGIN} & \text{A} = \text{A} + 100 \\
& & \text{B} = \text{B} - 100 & \text{COMMIT}
\end{align*}
\]

\[
\begin{align*}
T2 & : & \text{BEGIN} & \text{A} = \text{A} \times 1.06 \\
& & \text{B} = \text{B} \times 1.06 & \text{COMMIT}
\end{align*}
\]

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

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

```sql
BEGIN T1
| A = A + 100 |
| B = B - 100 |
COMMIT

BEGIN T2
| A = A * 1.06 |
| B = B * 1.06 |
COMMIT
```

Example

• Q: What are the possible outcomes of running T1 and T2 together?
• A: Many! But A+B should be:
  $2000 * 1.06 = \$2120$
• 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.

Example

• Legal outcomes:
  – A = 1166, B = 954 → $\$2120$
  – A = 1160, B = 960 → $\$2120$
• The outcome depends on whether T1 executes before T2 or vice versa.
Serial Execution Example

Interleaving Transactions

- We can also interleave the txns in order to maximize concurrency.
  - Slow disk/network I/O.
  - Multi-core CPUs.

Interleaving Example (Good)
Interleaving Example (Bad)

The bank lost $6!

Interleaving Example (Bad)

Schedule

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

DBMS’s View

Correctness

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

• **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.

Formal Properties of Schedules

• **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.
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
```

Read-Write Conflicts

- Unrepeatable Reads
Write-Write Conflicts

- Overwriting Uncommitted Data

Solution

- **Q:** How could you guarantee that all resulting schedules are correct (i.e., serializable)?
- **A:** Use locks!

Executing without Locks
Executing with Locks

• **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)

<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...*

Concurrency 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)

*We will discuss T/O methods in future classes.*
Two-Phase Locking

- **Phase 1: Growing**
  - Each txn requests the locks that it needs from the DBMS’s lock manager.
  - The lock manager grants/denies lock requests.

- **Phase 2: Shrinking**
  - The txn is allowed to only release locks that it previously acquired. It cannot acquire new locks.

- The txn is not allowed to acquire/upgrade locks after the growing phase finishes.

Transaction Lifetime

- **Growing Phase**
- **Shrinking Phase**

2PL Violation!
Executing with 2PL

2PL Observations

- There are schedules that are serializable but would not be allowed by 2PL.
- Locking limits concurrency.
- May lead to deadlocks.
- May still have “dirty reads”
  - Solution: **Strict 2PL**

Strict Two-Phase Locking

- A schedule is **strict** if a value written by a txn is not read or overwritten by other txns until that txn finishes.
- Advantages:
  - Recoverable.
  - Do not require cascading aborts.
  - Aborted txns can be undone by just restoring original values of modified tuples.
Strict Two-Phase Locking

- Txns hold all of their locks until commit.
- Good:
  - Avoids “dirty reads” etc
- Bad:
  - Limits concurrency even more
  - And still may lead to deadlocks

The txn is not allowed to acquire/upgrade locks after the growing phase finishes.

Transaction Lifetime

Growing Phase  Shrinking Phase

Q: Why is avoiding “dirty reads” important?
Strict Two-Phase Locking

• **Q:** Why is avoiding “dirty reads” important?
• **A:** If a txn aborts, all actions must be undone. Any txn that read modified data must also be aborted.

Locking in Practice

• You typically don’t set locks manually.
• Sometimes you will need to provide the DBMS with hints to help it to improve concurrency.
• Also useful for doing major changes.

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!

Transaction Durability

Buffer Pool

Memory

Disk

BEGIN
R(A)
W(A)
\vdots
COMMIT

T1

Buffer 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?

Write-Ahead Log

• Log record format:
  – <txnId, objectId, beforeValue, afterValue>
  – Each transaction writes a log record first, before doing the change
• When a txn finishes, the DBMS will:
  – Write a <commit> record on the log
  – Make sure that all log records are flushed before it returns an acknowledgement to application.

Write-Ahead Log

• After a failure, DBMS “replays” the log:
  – Undo uncommitted transactions
  – Redo the committed ones
Write-Ahead Log

T1
BEGIN
W(A)
W(B)
COMMIT

The DBMS hasn’t flushed memory to disk at this point.

Safe to return result to application.

CRASH!

W(A)
W(B)
⋮

COMMIT

The DBMS hasn’t flushed memory to disk at this point.

Before Value

After Value

W(A)
W(B)
⋮

COMMIT

Write-Ahead Log

T1
BEGIN
W(A)
W(B)
COMMIT

CRASH!

We have to undo T1!

D

We have to redo T1!

D

Recovering After a Crash

• At the end – all committed updates and only those updates are reflected in the database.

• Some care must be taken to handle the case of a crash occurring during the recovery process!
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.

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.
Summary

• 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

Recovery

Concurrency Control