Introduction to Database Systems
CSE 444
Lecture #11
Feb 12 2001

Announcements

HW#2 due on Wed
MidTerm will be returned next week

Concurrency Control: Review

- Provides Isolation
- Correctness = Serializability
- Stronger Condition: Conflict Serializability
  - Tested through precedence graph
- Implemented through locking
  - Compatibility among locking modes
  - Locking Protocol: 2PL

The Phantom Problem

Accounts: {(1, Redmond, 100), (2, Redmond, 40, (3, UW, 1000)}
Assets: {(Redmond, 140), (UW, 1000)}

T1: Add all accounts in Redmond and compare to total in assets. Report error
T2: Insert a new account {(7, Redmond, 5000)}

Phantom Problem: Analysis

T1 locks all existing Redmond accounts and reads accounts
T2 locks and introduces the new account and assets. Releases all locks
T1 locks the assets data and compares total
Schedule is not serial
- The new account is a phantom tuple

Observation
- Ensure that the "right" objects are locked
  - Lock all accounts with branch = Redmond
  - No change in 2PL needed

Implementing Locking

- Needs to execute Lock and Unlock as atomic operations
- Needs to be very fast ~100 instructions
- Lock Table
  - Low-level data structure in memory (not SQL Table!)
  - Implemented as a hash table
Issues in Managing Locks

- Multi-granularity locking
  - Concurrency v.s. locking overhead
  - Intention locks on higher-level objects
  - Lock Escalation
- Hot spots
  - Minimize lock duration

SQL-92 Syntax for Transactions

- **Start** Transaction: No explicit statement. Implicitly started
  - By a SQL statement
  - TP monitor (agents other than application programs)
- **End** Transaction:
  - By COMMIT or ROLLBACK
  - By external agents

SQL-92: Setting the Properties of Transactions

- **SET TRANSACTION**
  - [READ ONLY | READ WRITE]
  - ISOLATION LEVEL
    - [READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ | READ COMMITTED]

Explanation of Isolation Levels

- **Read Uncommitted**
  - Can see uncommitted changes of other transactions
  - Dirty Read, Unrepeatable Read
  - Recommended only for statistical functions
- **Read Committed**
  - Can see committed changes of other transactions
  - No Dirty read, but unrepeatable read possible
  - Acceptable for query/decision-support
- **Repeatable Read**
  - No dirty or unrepeatable read
  - May exhibit phantom phenomenon
- **Serializable**

Implementation of Isolation Levels

<table>
<thead>
<tr>
<th>ISOLATION LEVEL</th>
<th>DIRTY READ</th>
<th>UNREPEATABLE READ</th>
<th>PHANTOM</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>No S locks; writers must run at higher levels</td>
</tr>
<tr>
<td>Read Committed</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Serial 2PL X locks; S locks released anytime</td>
</tr>
<tr>
<td>Repeatable Reads</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Serial 2PL on data</td>
</tr>
<tr>
<td>Serializable</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Serial 2PL on data and indices (or predicate locking)</td>
</tr>
</tbody>
</table>

Summary of Concurrency Control

- Concurrency control key to a DBMS.
- Transactions and the ACID properties:
  - I handled by concurrency control.
  - A & D coming soon with logging & recovery.
- Conflicts arise when two Xacts access the same object, and one of the Xacts is modifying it.
- Serial execution is our model of correctness.
Summary of Concurrency Control (Contd.)

- Serializability allows us to “simulate” serial execution with better performance.
- 2PL: A simple mechanism to get serializability.
  - Lock manager module automates 2PL
    - Lock table is a big main-mem hash table
- Deadlocks are possible, and typically a deadlock detector is used to solve the problem.

Recovery

Reading: Chapter 8

Review: The ACID properties

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

The Recovery Manager guarantees Atomicity & Durability.

Motivation

- Atomicity:
  - Transactions may abort (“Rollback”).
- Durability:
  - What if DBMS stops running? (Causes?)
  - Desired Behavior after system restarts:
    - T1, T2 & T3 should be durable.
    - T4 & T5 should be aborted (effects not seen).

Rollback and Concurrency

- How does one undo the effects of a xact?
- What if another Xact sees these effects??
  - Must undo that Xact as well

Cascading Aborts

- Abort of T1 requires abort of T2!
  - Cascading Abort
- An ACA (avoids cascading abort) schedule is one in which cascading abort cannot arise:
  - A Xact only reads data from committed Xacts.
Recoverable Schedules

Abort of T1 requires abort of T2!
But T2 has already committed!
A recoverable schedule is one in which this cannot happen.
 préc.: a Xact commits only after all the Xacts it reads from commit.
ACA implies recoverable (but not vice-versa!).
2PL ensure that only recoverable schedules arise

What is Recovery?

Concurrency control is in effect.
Strict 2PL, in particular
Discussion on Recovery may be based on
Single user, but multiple concurrent transactions
User does transactions but failures are possible
Recovery: scheme to guarantee Atomicity & Durability of user transactions

Assumption (for Simplicity)

Page Granularity for everything
- Database = Set of Pages
- Each update by a transaction applies to only one page
- Each update writes a whole page
- Locks are set on pages

Storage Model

Stable Database
- One copy for every database page
Database Buffer/Cache
- One copy of some of the database pages accessed/updated
- May contain updates that have not been written to stable database): dirty pages

Storage Model: Cache Manager

Cache Descriptor Table
- Database Page
- Main memory address
- Dirty bit
- Pin count
Operations
- Fetch(P), Pin(P), UnPin(P)
- Flush(P) [sync write], Deallocate(P)

A Simplified Way of Thinking

INPUT(X): read element X to memory buffer
READ(X,t): copy element X to transaction local variable t
WRITE(X,t): copy transaction local variable t to element X
OUTPUT(X): write element X to disk
Somewhat inaccurate account?
Example
READ(A,t); t := t*2; WRITE(A,t)
READ(B,t); t := t*2; WRITE(B,t)

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Types of Failures

- **Data Integrity**
  - Prevent by constraints in the database/good software practices, Fix with data cleaning applications
  - Fix with recovery

- **Transaction failure: When a transaction aborts**
  - Fix with recovery

- **System failures: Loss of contents of volatile store (Power/OS outrage)**
  - Prevent by stable storage, Fix with recovery

- **Media Failure: Loss of contents of disk**
  - Prevent by using redundancy (RAID, archive), Fix with recovery

Handling System Failures

- When system crashes, internal state is lost
  - Don’t know which parts executed and which didn’t
- Remedy: use a log
  - A file that records every single update

The Log

- An append-only file containing log records
- Multiple transactions run concurrently, log records are interleaved
- After a system crash, use log to:
  - Undo other transactions that didn’t commit
  - Redo some transaction that didn’t commit
- Techniques
  - Undo Logging
  - Redo Logging
  - Undo/Redo Logging (preferred)

 Undo Logging

Log records
- `<START T>` = transaction T has begun
- `<COMMIT T>` = T has committed
- `<ABORT T>` = T has aborted
- `<T,X,v>` = T has updated element (page) X, and its old value was v

Undo-Logging Rules

U1: If T modifies X, then the log record `<T,X,v>` must be written to disk before X is written to disk
U2: If T commits, then `<COMMIT T>` must be written to log only after all changes by T are written to disk
### Recovery with Undo Log

#### Idea 1. Decide for each transaction T whether it is completed or not
- `<START T>….<COMMIT T>….    = yes`
- `<START T>….<ABORT T>…….    = yes`
- `<START T>………………………   = no`

#### Idea 2. Undo all modifications by incomplete transactions

---

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td><code>&lt;START T&gt;</code></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td><code>&lt;COMMIT T&gt;</code></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td><code>&lt;T,A,t&gt;</code></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td><code>&lt;T,B,t&gt;</code></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td><code>&lt;T,A,t&gt;</code></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td><code>&lt;T,B,t&gt;</code></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td><code>&lt;T,A,t&gt;</code></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td><code>&lt;T,B,t&gt;</code></td>
</tr>
</tbody>
</table>

---

#### Note: all undo commands are idempotent
- If we perform them a second time, no harm is done
- E.g., if there is a system crash during recovery, simply restart recovery from scratch

---

#### When do we stop reading the log?
- We cannot stop until we reach the beginning of the log file
- This is impractical
- Better idea: use checkpointing
**Checkpointing**

Checkpoint the database periodically
- Stop accepting new transactions
- Wait until all current transactions complete
- Flush dirty pages to disk
- Write a <CKPT> log record
- Resume transactions

**Nonquiescent Checkpointing**

- Problem with checkpointing: database freezes during checkpoint
- Would like to checkpoint while database is operational
- = nonquiescent (fuzzy) checkpointing

**Undo Recovery with Checkpointing**

During recovery,
- Can stop at first <CKPT>
- Other transactions

**Nonquiescent Checkpointing**

- Stop accepting any new update/commit/abort
  - Make a list of all dirty pages in the buffer
  - Write a <START CKPT(T1,...,Tk)> where T1,...,Tk are all active transactions
- Start normal operation
  - Flush unpinned dirty pages as a low-priority item
- When all of T1,...,Tk have completed, and their dirty pages written out
  - Write <END CKPT>
  - Cannot start a <START CKPT...> until earlier <END CKPT> is complete

**Undo Recovery with Nonquiescent Checkpointing**

During recovery,
- Can stop at first <START CKPT>
- Q: What if no <End CKPT> in the log?

**Redo Logging**

Log records
- <START T> = transaction T has begun
- <COMMIT T> = T has committed
- <ABORT T> = T has aborted
- <T,X,v> = T has updated element X, and its new value is v
Redo-Logging Rules

R1: If T modifies X, then both <T,X,v> and <COMMIT T> must be written to log before X is written (flushed) to disk.

Lazy write to disk – may need to “redo” work during recovery.

Recovery with Redo Log

After system’s crash, run recovery manager

Step 1. Decide for each transaction T whether it is completed or not

- <START T>….<COMMIT T>…. = yes
- <START T>….<ABORT T>…. = yes
- <START T>......................... = no

Step 2. Read log from the beginning, redo all updates of committed transactions.

Recovery using Redo Log

- For committed transactions
  - Replay Write() for the log record <T,X,v>
- For each incomplete transaction T
  - Write <Abort T> to log
- Follow Example 8.8

Example: Recovery with Redo Log

```
<START T1>
<T1,X1,v1>
<START T2>
<T2,X2,v2>
<START T3>
<T3,X3,v3>
<COMMIT T2>
<T3,X4,v4>
<T3,T3,v3>
```

Nonquiescent Checkpointing

- Write a <START CKPT(T1,...,Tk)>
  - where T1,...,Tk are all active transactions
- Flush to disk all blocks of committed transactions (dirty blocks), while continuing normal operation
- When all blocks have been written, write <END CKPT>
**Redo Recovery with Nonquiescent Checkpointing**

**Step 1:** look for
*The last*

<END CKPT>

---

**Step 2:** redo from there, ignoring transactions committed earlier

---

All OUTPUTs of T1 are known to be on disk

---

**Comparison Undo/Redo**

%%Undo logging:
- OUTPUT must be done early
- If <COMMIT T> is seen, T definitely has written all its data to disk

%%Redo logging
- OUTPUT must be done late
- If <COMMIT T> is not seen, T definitely has not written any of its data to disk

---

**Undo/Redo Logging**

%%Log Record: \( <T, X, u, v> \) = T has updated element \( X \), its *old* value was \( u \), and its *new* value is \( v \)

%%Rule: If T modifies \( X \), then the log record \( <T, X, u, v> \) must be written to disk before \( X \) is written to disk

---

**Recovery with Undo/Redo Log**

After system’s crash, run recovery manager

%%Redo all committed transaction beginning at last checkpoint

%%Undo all uncommitted transactions, until last checkpoint

---

**Recovery with Redo Log**

---
Media Failure

Redundancy is the key
- Shadowed Disk/RAID either for database or at least for the log
- Cannot afford to lose part of a log!
- Only place which has before-image (after-image) of uncommitted data written (not written) to disk
- Minimize shared hardware
- Using Archive (next lecture)

Summary

- Checkpointing: A quick way to limit the amount of log to scan on recovery.
- Recovery works in 3 phases:
  - Analysis: Forward from checkpoint.
  - Redo: Forward from checkpoint.
  - Undo: Backward until checkpoint
- Tolerating media Failure requires more redundancy
- Many more optimizations in real system