Lecture 15: Data Storage, Recovery

Monday, February 13, 2006

Outline

• Disks 11.3
  – Recommended reading: entire chapter 11
• Recovery using undo logging 17.2
The Mechanics of Disk

Mechanical characteristics:
• Rotation speed (5400RPM)
• Number of platters (1-30)
• Number of tracks (<=10000)
• Number of bytes/track(10^5)

Unit of read or write: **disk block**
Once in memory: **page**
Typically: 4k or 8k or 16k

Disk Access Characteristics

• **Disk latency** = time between when command is issued and when data is in memory

• Disk latency = seek time + rotational latency
  – Seek time = time for the head to reach cylinder
    • 10ms – 40ms
  – Rotational latency = time for the sector to rotate
    • Rotation time = 10ms
    • Average latency = 10ms/2
• Transfer time = typically 40MB/s
• Disks read/write one block at a time
**RAID**

Several disks that work in parallel
- Redundancy: use parity to recover from disk failure
- Speed: read from several disks at once

Various configurations (called *levels*):
- RAID 1 = mirror
- RAID 4 = $n$ disks + 1 parity disk
- RAID 5 = $n+1$ disks, assign parity blocks round robin
- RAID 6 = “Hamming codes”

**Buffer Management in a DBMS**

Page Requests from Higher Levels

- Data must be in RAM for DBMS to operate on it!
- Table of <frame#, pageid> pairs is maintained
Buffer Manager

Needs to decide on page replacement policy

• LRU
• Clock algorithm

Both work well in OS, but not always in DB

Enables the higher levels of the DBMS to assume that the needed data is in main memory.

Least Recently Used (LRU)

• Order pages by the time of last accessed
• Always replace the least recently accessed

<table>
<thead>
<tr>
<th>P5, P2, P8, P4, P1, P9, P6, P3, P7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access P6</td>
</tr>
</tbody>
</table>

| P6, P5, P2, P8, P4, P1, P9, P3, P7 |

LRU is expensive (why ?); the clock algorithm is good approx
Buffer Manager

Why not use the Operating System for the task??

- DBMS may be able to anticipate access patterns
- Hence, may also be able to perform prefetching
- DBMS needs the ability to force pages to disk, for recovery purposes
- need fine grained control for transactions

Transaction Management

Two parts:

• Recovery from crashes:  ACID
• Concurrency control:    ACID

Both operate on the buffer pool
Recovery

From which of the events below can a database actually recover?

- Wrong data entry
- Disk failure
- Fire / earthquake / bankruptcy / ….
- Systems crashes

Recovery

<table>
<thead>
<tr>
<th>Type of Crash</th>
<th>Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrong data entry</td>
<td>Constraints and Data cleaning</td>
</tr>
<tr>
<td>Disk crashes</td>
<td>Redundancy: e.g. RAID, archive</td>
</tr>
<tr>
<td>Fire, theft, bankruptcy…</td>
<td>Buy insurance, Change jobs…</td>
</tr>
<tr>
<td>System failures: e.g. power</td>
<td>DATABASE RECOVERY</td>
</tr>
</tbody>
</table>
System Failures

• Each transaction has *internal state*
• 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 action of the transaction

Transactions

• Assumption: the database is composed of **elements**
  – Usually 1 element = 1 block
  – Can be smaller (=1 record) or larger (=1 relation)
• Assumption: each transaction reads/writes some elements
Primitive Operations of Transactions

- **READ(X,t)**
  - copy element X to transaction local variable t
- **WRITE(X,t)**
  - copy transaction local variable t to element X
- **INPUT(X)**
  - read element X to memory buffer
- **OUTPUT(X)**
  - write element X to disk

Example

```
START TRANSACTION
READ(A,t);
t := t*2;
WRITE(A,t);
READ(B,t);
t := t*2;
WRITE(B,t)
COMMIT;
```

Atomicity:
BOTH A and B are multiplied by 2
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>INPUT(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<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>INPUT(B)</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>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</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>

Crash occurs after OUTPUT(A), before OUTPUT(B)
We lose atomicity
The Log

• An append-only file containing log records
• Note: multiple transactions run concurrently, log records are interleaved
• After a system crash, use log to:
  – Redo some transaction that didn’t commit
  – Undo other transactions that didn’t commit
• Three kinds of logs: undo, redo, undo/redo

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 X, and its old value was v
### Table 1

<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>INPUT(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;COMMIT T&gt;</td>
</tr>
</tbody>
</table>

### Table 2

<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>INPUT(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;COMMIT T&gt;</td>
</tr>
</tbody>
</table>

**Crash!**

**WHAT DO WE DO?**
**After Crash**

- **In the first example:**
  - We UNDO both changes: A=8, B=8
  - The transaction is atomic, since none of its actions has been executed

- **In the second example**
  - We don’t undo anything
  - The transaction is atomic, since both it’s actions have been executed
Undo-Logging Rules

U1: If T modifies X, then \(<T,X,v>\) must be written to disk before OUTPUT(X)

U2: If T commits, then OUTPUT(X) must be written to disk before \(<\text{COMMIT } T>\)

• Hence: OUTPUTs are done \textit{early}, before the transaction commits
Recovery with Undo Log

After system’s crash, run recovery manager

• 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
Recovery with Undo Log

... ...
<T6,X6,v6>
...

<START T5>
<START T4>
<T1,X1,v1>
<T5,X5,v5>
<T4,X4,v4>
<COMMIT T5>
<T3,X3,v3>
<T2,X2,v2>

Question 1 in class:
Which updates are undone?

Question 2 in class:
How far back do we need to read in the log?

• 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
Recovery with Undo Log

When do we stop reading the log?
• We cannot stop until we reach the beginning of the log file
• This is impractical

Instead: use checkpointing