**Introduction to Database Systems**

CSE 444

Lecture #12
Feb 14 2001

**Announcements**

- HW#2 due today
- MidTerm will be returned next Wed

---

**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

Step 1: look for the last <END CKPT>
Step 2: redo from there, ignoring transactions committed earlier

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
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
Archive: Fuzzy Dump

<Begin Dump>
<br>Start Ckpt (T1, T2)>
<T1, A, 1, 5>
<T2, C, 3, 6>
<T1, B, 2, 7>
<Commit T2>
<End Ckpt>
<End Dump>

Archive: Pragmatics

<Usually a separate media recovery log
<br>Disk Contention
<Media Log Archiver read from the head
<Log is aepnd-only
<Use two pairs of shadowed log disks
<Avoid keeping undo information in media recovery log
<Archive only when their entire content is committed
<Use write-lock on pages

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

Storage

Reading: Chapter 3, 4

Memory Hierarchy

Typical storage hierarchy:
- Main memory (RAM) for currently used data.
- Disk for the main database (secondary storage).
- Tapes for archiving older versions of the data (tertiary storage).
This has major implications for DBMS design!
- READ: transfer data from disk to main memory (RAM).
- WRITE: transfer data from RAM to disk.
Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

Disks

Secondary storage device of choice.
Main advantage over tapes: random access vs. sequential.
Data is stored and retrieved in units called disk blocks or pages.
Unlike RAM, time to retrieve a disk page varies depending upon location on disk.
Therefore, relative placement of pages on disk has major impact on DBMS performance!
Components of a Disk

The platters spin (say, 100rps). The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a cylinder (imaginary!). Only one head reads/writes at any one time.

- **Block size** is a multiple of sector size (which is fixed).

Accessing a Disk Page

- Time to access (read/write) a disk block:
  - **Seek time** (moving arms to position disk head on track)
  - **Rotational delay** (waiting for block to rotate under head)
  - Often called "rotational latency"
  - **Transfer time** (actually moving data to/from disk surface)

Seek time and rotational delay dominate.

- Seek time varies from about 1 to 20msec
- Rotational delay varies from 0 to 10msec
- Transfer rate is about 1msec per 4KB page

Key to lower I/O cost: reduce seek/rotation delays! Hardware vs. software solutions?

Arranging Pages on Disk

- "Next" block concept:
  - Blocks on same track, followed by
  - Blocks on same cylinder, followed by
  - Blocks on adjacent cylinder
- Blocks in a file should be arranged sequentially on disk (by "next"), to minimize seek and rotational delay.
- For a sequential scan, **pre-fetching** several pages at a time is a big win!

Disk Space Management

- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
  - Allocate/de-allocate a page
  - Read/write a page
- One such "higher level" is the buffer manager, which receives a request to bring a page into memory and then, if needed, requests the disk space layer to read the page into the buffer pool.

Files of Records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on records, and files of records.
- **FILE**: A collection of pages, each containing a collection of records. Must support:
  - Insert/delete/modify records
  - Read a particular record (specified using record id)
  - Scan all records (possibly with some conditions on the records to be retrieved)
Record Formats: Fixed Length

- Information about field types same for all records in a file; stored in system catalogs.
- Finding i'th field requires scan of record.
- Note the importance of schema information!

```
<table>
<thead>
<tr>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>L2</td>
<td>L3</td>
<td>L4</td>
</tr>
</tbody>
</table>
```

Base address (B) Address = B+L1+L2

Record Header

- Need the header because:
  - The schema may change for a while new+old may coexist
  - Records from different relations may coexist

```
<table>
<thead>
<tr>
<th>To schema length</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
</tr>
</thead>
<tbody>
<tr>
<td>header timestamp</td>
<td>L1</td>
<td>L2</td>
<td>L3</td>
<td>L4</td>
</tr>
</tbody>
</table>
```

Variable Length Records

- Place the fixed fields first: F1, F2
- Then the variable length fields: F3, F4
- Null values take 2 bytes only
- Sometimes they take 0 bytes (when at the end)

Storing Records in Blocks

- Blocks have fixed size (typically 4k)

```
<table>
<thead>
<tr>
<th>BLOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
</tr>
<tr>
<td>R4</td>
</tr>
</tbody>
</table>
```

Spanning Records Across Blocks

- When records are very large
- Or even medium size: saves space in blocks

Modifications: Insertion

- File is unsorted: add it to the end (easy 😊)
- File is sorted:
  - Is there space in the right block?
    - Yes: we are lucky, store it there
  - Is there space in a neighboring block?
    - Look 1-2 blocks to the left/right, shift records
  - If anything else fails, create overflow block
Overflow Blocks

Block_{n-1}  Block_n  Block_{n+1}  Overflow

After a while the file starts being dominated by overflow blocks: time to reorganize

Modifications: Deletions

Free space in block, shift records
Maybe be able to eliminate an overflow block
Can never really eliminate the record, because others may point to it
Place a tombstone instead (a NULL record)

Modifications: Updates

If new record is shorter than previous, easy 😊
If it is longer, need to shift records, create overflow blocks

Physical Addresses

Each block and each record have a physical address that consists of:
The host
The disk
The cylinder number
The track number
The block within the track
For records: an offset in the block
Sometimes this is in the block's header

Logical Addresses

Logical address: a string of bytes (10-16)
More flexible: can blocks/records around
But need translation table:

<table>
<thead>
<tr>
<th>Logical address</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>P1</td>
</tr>
<tr>
<td>L2</td>
<td>P2</td>
</tr>
<tr>
<td>L3</td>
<td>P3</td>
</tr>
</tbody>
</table>

Main Memory Address

When the block is read in main memory, it receives a main memory address
Need another translation table

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Logical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>L1</td>
</tr>
<tr>
<td>M2</td>
<td>L2</td>
</tr>
<tr>
<td>M3</td>
<td>L3</td>
</tr>
</tbody>
</table>
**Optimization: Pointer Swizzling**

- The process of replacing a physical/logical pointer with a main memory pointer
- Still need translation table, but subsequent references are faster

**Indexes**

- An index on a file speeds up selections on the search key fields for the index.
- Any subset of the fields of a relation can be the search key for an index on the relation.
- Search key is not the same as key (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of data entries, and supports efficient retrieval of all data entries with a given key value $k$.

**Index Classification**

- Primary/secondary
- Clustered/unclustered
- Dense/sparse
- B+ tree / Hash table / ...

**Primary Index**

- File is sorted on the index attribute
- Dense index: sequence of (key,pointer) pairs

**Primary Index with Duplicate Keys**

- Sparse index

- Dense index:
Primary Index with Duplicate Keys

Sparse index: pointer to lowest search key in each block:

Search for 20

Primary Index with Duplicate Keys

Better: pointer to lowest new search key in each block:

Search for 20

Secondary Indexes

To index other attributes than primary key
Always dense (why?)

Clustered/Unclustered

Primary indexes = usually clustered
Secondary indexes = usually unclustered

Clustered vs. Unclustered Index

Secondary Indexes

Applications:
- Index other attributes than primary key
- Index unsorted files (heap files)
- Index clustered data
Applications of Secondary Indexes

Clustered data
Company(name, city), Product(pid, maker)

Select city
From Company, Product
Where name=maker
and pid="p045"

Select pid
From Company, Product
Where name=maker
and city="Seattle"

Composite Search Keys

Examples of composite key indexes using lexicographic order.

B+ Trees

Search trees
Idea in B Trees:
make 1 node = 1 block

Idea in B+ Trees:
Make leaves into a linked list (range queries are easier)

B+ Trees Basics

Parameter d = the degree
Each node has >= d and <= 2d keys
(except root)

Each leaf has >= d and <= 2d keys:

B+ Tree Example

B+ Tree Design

How large d ?
Example:
Key size = 4 bytes
Pointer size = 8 bytes
Block size = 4096 bytes

2d x 4 + (2d+1) x 8 <= 4096

d = 170