## Introduction to Database Systems

## CSE 444

**Lecture #12**
**Feb 14 2001**

## Announcements

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

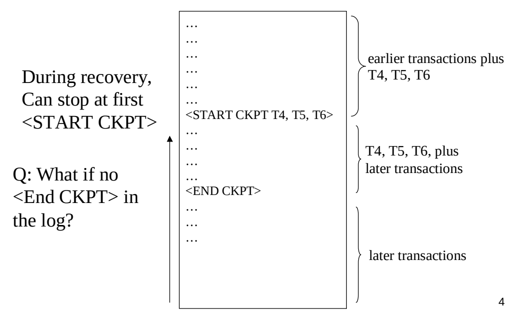
2

## 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

3

## Undo Recovery with Nonquiescent Checkpointing




4

## 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 <u>new</u> value is v

5

## Redo-Logging Rules

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

Lazy write to disk – may need to "redo" work during recovery

6

| Action | T | Mem A | Mem B | Disk A | Disk B | Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| REAT(A,t) | 8 | 8 | | 8 | 8 | |
| t:=t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,16> |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,16> |
| | | | | | | <COMMIT T> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |

7

## 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

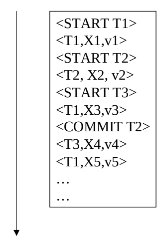
8

## 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

9

## Example: Recovery with Redo Log



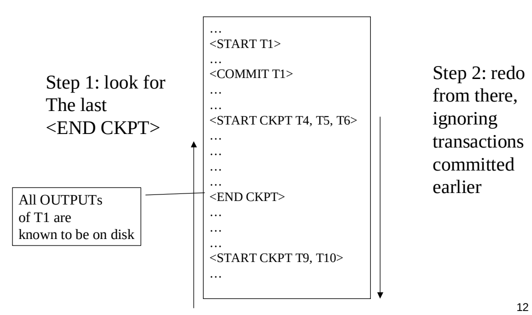

10

## 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>

11

## Redo Recovery with Nonquiescent Checkpointing




12

## 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

13

## Undo/Redo Logging

- <u>Log Record</u>: <T,X,u,v>= T has updated element X, its *old* value was u, and its *new* value is v
- <u>Rule:</u> If T modifies X, then the <u>log</u> record <T,X,u,v> must be written to disk <u>before</u> X is written to disk

14

| Action | T | Mem A | Mem B | Disk A | Disk B | Log |
|---|---|---|---|---|---|---|
| | | | | | | <START T> |
| REAT(A,t) | 8 | 8 | | 8 | 8 | |
| t:=t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8,16> |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t:=t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8,16> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| | | | | | | <COMMIT T> |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |

15

## 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

16

## Recovery with Redo Log

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

17

## 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

18

## Archive: Fuzzy Dump

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

19

## Archive: Pragmatics

- Usually a separate media recovery log
- Disk Contention
  - Media Log Archiver read from the head
  - Log is apepnd-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

20

## 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

21

## 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!

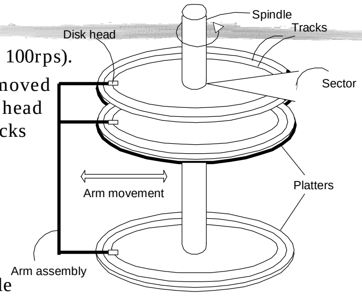
## Components of a Disk

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?

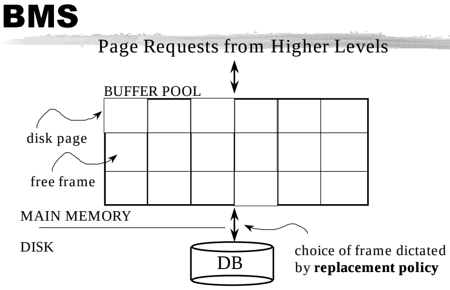
## Arranging Pages on Disk

- \`*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, <u>*pre-fetching*</u> 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.

## Buffer Management in a DBMS

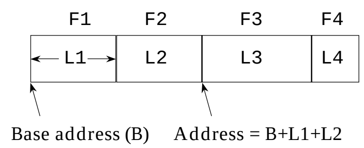



- |*Table of <frame#, pageid> pairs is maintained.*

## Files of Records

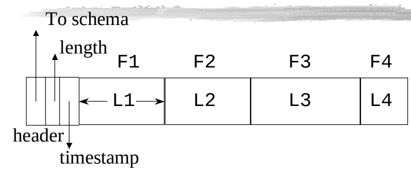
- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- <u>FILE</u>: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan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

## Record Formats: Fixed Length



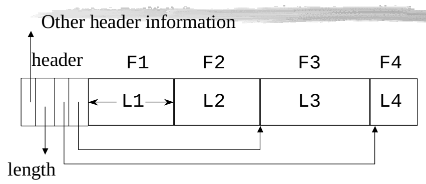

⌘Information about field types same for all records in a file; stored in *system catalogs.*
⌘Finding *i'th* field requires scan of record.
⌘**Note the importance of schema information!**

## Record Header




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

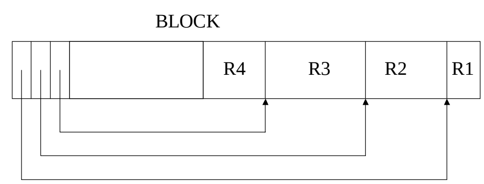
## Variable Length Records




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)

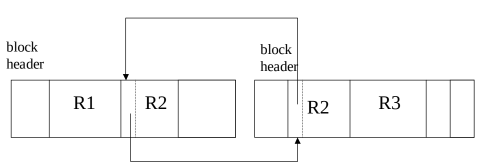
## Storing Records in Blocks

⌘Blocks have fixed size (typically 4k)




34

## Spanning Records Across Blocks




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

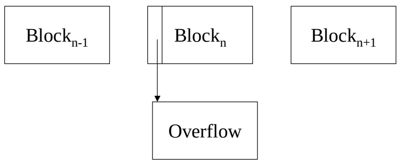
35

## 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*

36

## Overflow Blocks




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

37

## 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)

38

## Modifications: Updates

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

39

## 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

40

## Logical Addresses

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

| Logical address | Physical address |
|---|---|
| L1 | P1 |
| L2 | P2 |
| L3 | P3 |

41

## Main Memory Address

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

| Memory address | Logical address |
|---|---|
| M1 | L1 |
| M2 | L2 |
| M3 | L3 |

42

## 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

43

## 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**.

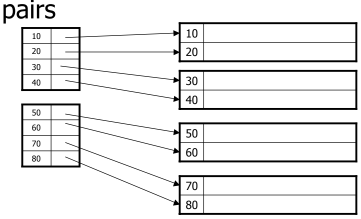
## Index Classification

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

45

## Primary Index

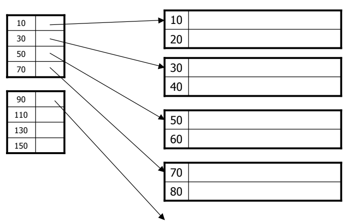
- File is sorted on the index attribute
- *Dense* index: sequence of (key,pointer) pairs




46

## Primary Index

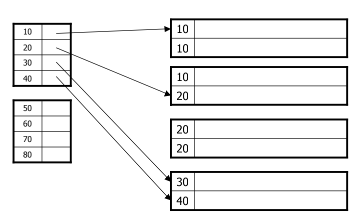
- *Sparse* index




47

## Primary Index with Duplicate Keys

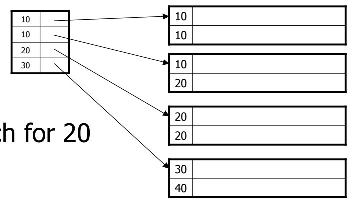
- Dense index:




48

## Primary Index with Duplicate Keys

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



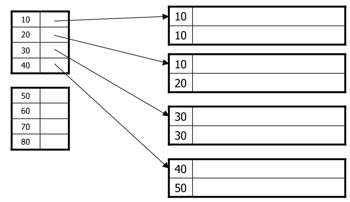

⌘Search for 20

49

## Primary Index with Duplicate Keys

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

⌘Search for 20

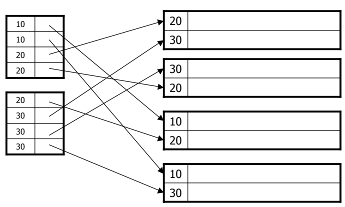



50

## Secondary Indexes

⌘To index other attributes than primary key

⌘Always dense (why ?)




51

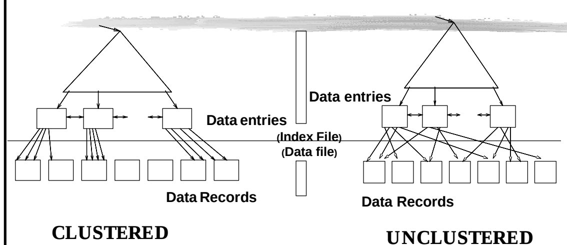
## Clustered/Unclustered

⌘Primary indexes = usually clustered

⌘Secondary indexes = usually unclustered

52

## Clustered vs. Unclustered Index




## Secondary Indexes

⌘Applications:

- index other attributes than primary key
- index unsorted files (heap files)
- index clustered data

54

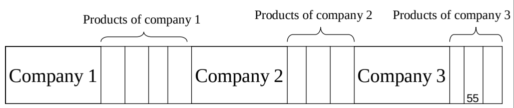
## Applications of Secondary Indexes

⌘*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"

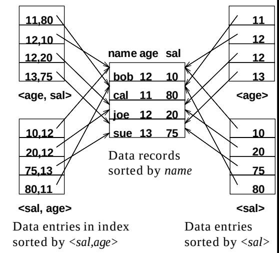





## Composite Search Keys

⌘ *Composite Search Keys*: Search on a combination of fields.
- Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
  - age=20 and sal =75
- Range query: Some field value is not a constant. E.g.:
  - age =20; or age=20 and sal > 10

Examples of composite key indexes using lexicographic order.




## B+ Trees

⌘Search trees

⌘Idea in B Trees:
- make 1 node = 1 block

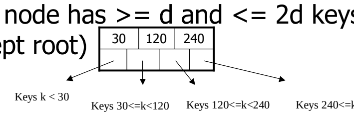
⌘Idea in B+ Trees:
- Make leaves into a linked list (range queries are easier)



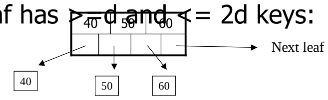
## B+ Trees Basics

⌘Parameter d = the *degree*

⌘Each node has >= d and <= 2d keys (except root)



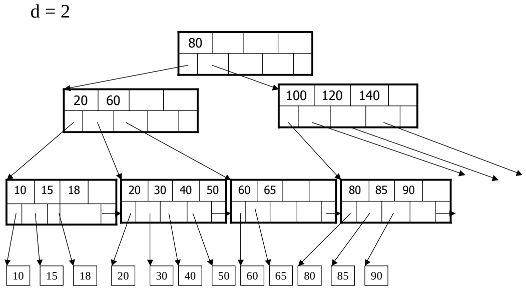

⌘Each leaf has >=d and <= 2d keys:






## B+ Tree Example

d = 2






## B+ Tree Design

⌘How large d ?

⌘Example:
- Key size = 4 bytes
- Pointer size = 8 bytes
- Block size = 4096 byes

⌘2d x 4 + (2d+1) x 8 <= 4096

⌘d = 170