Lecture 15: Data Storage

Tuesday, February 20, 2001

Outline

- Storing Data: Disks (7.1-7.3)
- Buffer manager (7.4)
- Representing data (7.5-7.7)
- External Sorting (Chapter 11)

The Memory Hierarchy

Processor Cache:
- access time 10 nano's

Main Memory = Disk Cache
- Volatile
- 256M-1G
- expensive
- Access time: 10-100 nanoseconds

Disk
- Persistent
- 2-10 GB storage
- speed:
  - Rate=5-10 MB/S
  - Access time= 10-15 msecs.

Tape
- 1.5 MB/S transfer rate
- 280 GB typical capacity
- Only sequential access
- Not for operational data

Main Memory

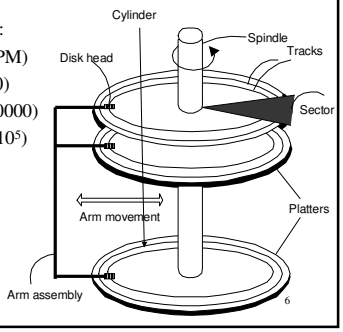
- Fastest, most expensive
- Today: 256MB are common even on PCs
- Many databases could fit in memory
  - New industry trend: Main Memory Database
  - E.g TimesTen
- Main issue is volatility

Secondary Storage

- Disks
- Slower, cheaper than main memory
- Persistent !!!
- The unit of disk I/O = ***block***
  - Typically 1 block = 4k
- Used with a main memory buffer

The Mechanics of Disk

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



Important 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 5-10MB/s
- Disks read/write one block at a time (typically 4kB)

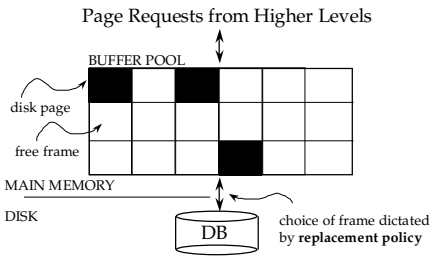
7

RAIDs

- = "Redundant Array of Independent Disks"
  - Was "inexpensive" disks
- Idea: use more disks, increase reliability
- Recall:
  - Database *recovery* helps after a systems crash, not after a disk crash
- 6 ways to use RAIDs. More important:
  - Level 4: use N-1 data disks, plus one parity disk
  - Level 5: same, but alternate which disk is the parity
  - Level 6: use Hamming codes instead of parity

8

Buffer Management in a DBMS



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

9

Buffer Manager

- When a page is first requested, it is read in a free frame
- The DBMS typically requests it to be *pinned*:
  - pin_count = how many processes requested it pinned
- When the DBMS writes to it, the buffer manager marks it *dirty*
- When the DBMS doesn't need it any more, *un-pinned*:
  - pin_count is decremented
- Typical replacement policy (always chooses among *un-pinned* frames):
  - LRU
  - Clock
  - MRU

10

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.

11

Representing Data Elements

- Relational database elements:

```
CREATE TABLE Product (
    pid INT PRIMARY KEY,
    name CHAR(20),
    description VARCHAR(200),
    maker CHAR(10) REFERENCES Company(name))
```

- A tuple is represented as a record

12

Representing Data Elements

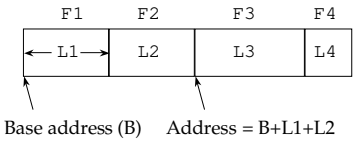
- Representing objects:

```
interface Company {
  attribute string name;
  relationship Set<Product> makes
               inverse Product::maker;
}
```

- An object is represented as a record plus object identifier
  - What to do with repeating fields (e.g. makes)

13

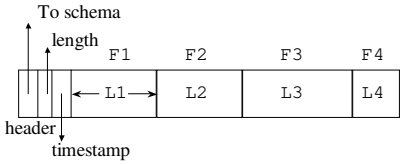
Record Formats: Fixed Length



- 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!**

14

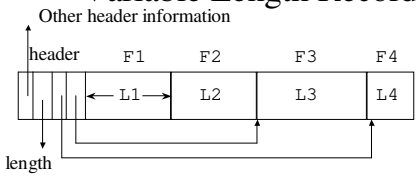
Record Header



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

15

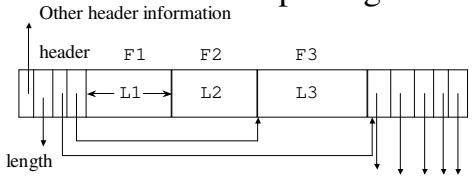
Variable Length Records



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)

16

Records With Repeating Fields

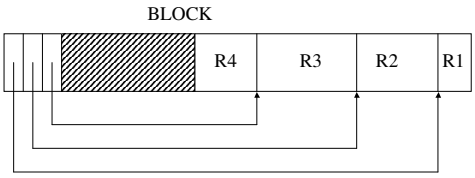


E.g. to represent one-many or many-many relationships

17

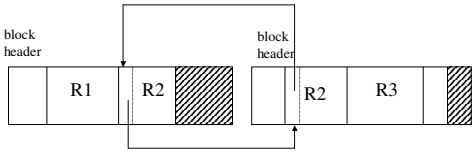
Storing Records in Blocks

- Blocks have fixed size (typically 4k)



18

Spanning Records Across Blocks



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

19

BLOB

- Binary large objects
- Supported by modern database systems
- E.g. images, sounds, etc.
- Storage: attempt to cluster blocks together

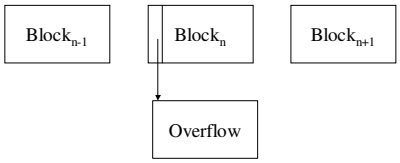
20

Modifications: Insertion

- File is unsorted (= ***heap file***)
  - 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*

21

Overflow Blocks



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

22

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)

23

Modifications: Updates

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

24

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

25

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

26

Main Memory Address

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

Memory address	Logical address
M1	L1
M2	L2
M3	L3

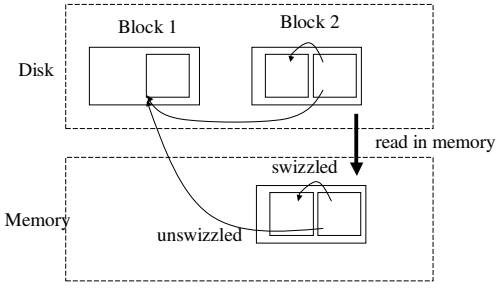
27

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

28

Pointer Swizzling



29

Pointer Swizzling

- *Automatic*: when block is read in main memory, swizzle all pointers in the block
- *On demand*: swizzle only when user requests
- *No swizzling*: always use translation table

30

Pointer Swizzling

- When blocks return to disk: pointers need unswizzled
- Danger: someone else may point to this block
  - Pinned blocks: we don't allow it to return to disk
  - Keep a list of references to this block

31

The I/O Model of Computation

- In main memory algorithms we care about CPU time
- In databases time is dominated by I/O cost
- Assumption: cost is given only by I/O
- Consequence: need to redesign certain algorithms
- Will illustrate here with sorting

32

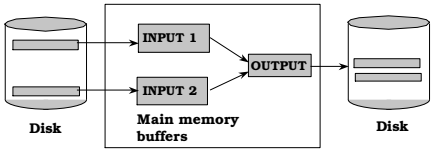
Sorting

- Illustrates the difference in algorithm design when your data is not in main memory:
  - Problem: sort 1Gb of data with 1Mb of RAM.
- Arises in many places in database systems:
  - Data requested in sorted order (ORDER BY)
  - Needed for grouping operations
  - First step in sort-merge join algorithm
  - Duplicate removal
  - Bulk loading of B+-tree indexes.

33

2-Way Merge-sort: Requires 3 Buffers

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, …, etc.:
  - three buffer pages used.



34

Two-Way External Merge Sort

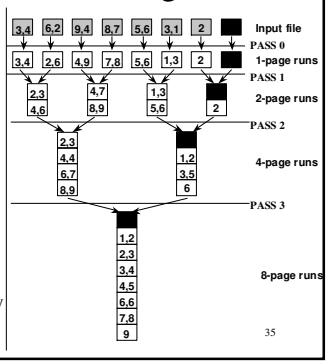
- Each pass we read + write each page in file.
- N pages in the file => the number of passes

$$= \lceil \log_2 N \rceil + 1$$

- So total cost is:

$$2N\left(\lceil \log_2 N \rceil + 1\right)$$

- Improvement: start with larger runs
- Sort 1GB with 1MB memory in 10 passes



35

Can We Do Better ?

- We have more main memory
- Should use it to improve performance

36

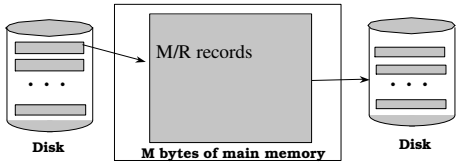
Cost Model for Our Analysis

- **B:** Block size
- **M:** Size of main memory
- **N:** Number of records in the file
- **R:** Size of one record

37

External Merge-Sort

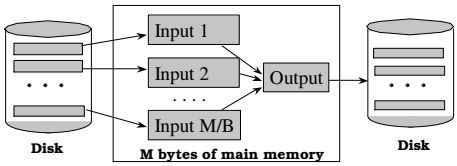
- Phase one: load M bytes in memory, sort
  - Result: runs of length M/R records



38

Phase Two

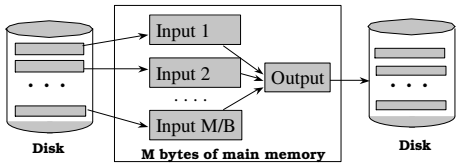
- Merge M/B – 1 runs into a new run
- Result: runs have now M/R (M/B – 1) records



39

Phase Three

- Merge M/B – 1 runs into a new run
- Result: runs have now M/R $(M/B – 1)^2$ records



40

Cost of External Merge Sort

- Number of passes: $1+\lceil \log_{M/B-1} \lceil NR/M \rceil \rceil$
- Think differently
  - Given B = 4KB, M = 64MB, R = 0.1KB
  - Pass 1: runs of length M/R = 640000
    - Have now sorted runs of 640000 records
  - Pass 2: runs increase by a factor of M/B – 1 = 16000
    - Have now sorted runs of 10,240,000,000 = 10^{10} records
  - Pass 3: runs increase by a factor of M/B – 1 = 16000
    - Have now sorted runs of 10^{14} records
    - Nobody has so much data !
- Can sort everything in 2 or 3 passes !

41