Lecture 20: Representing Data Elements

Wednesday, November 8, 2000

Outline

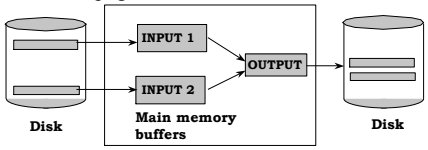
- External sorting (2.3.3-2.3.5)
- Representing data elements (3)

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.

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.



Two-Way External Merge Sort

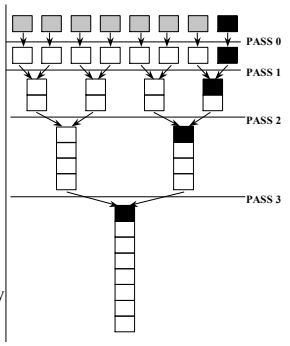
- Each pass we read + write each page in file.
- N pages in the file => the number of passes

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

- 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



Can We Do Better ?

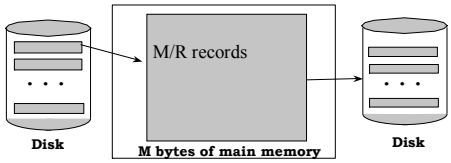
- We have more main memory
- Should use it to improve performance

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

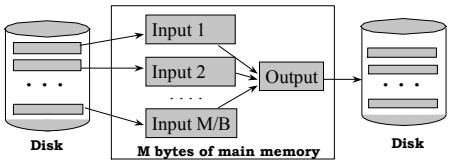
External Merge-Sort

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



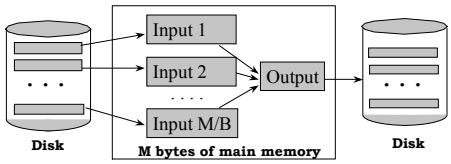
Phase Two

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



Phase Three

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



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 !

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

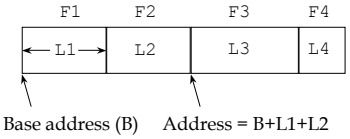
Representing Data Elements

- Representing objects:

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

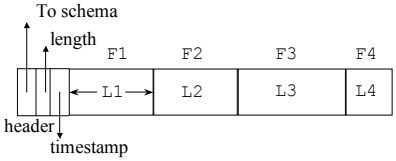
- An object is represented as a record plus object identifier
  - What to do with repeating fields (e.g. makes)

Record Formats: Fixed Length



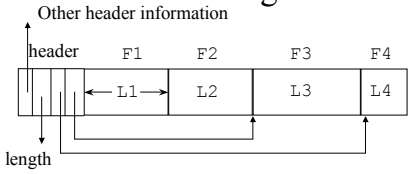
- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field requires scan of record.
- **Note the importance of schema information!**

Record Header



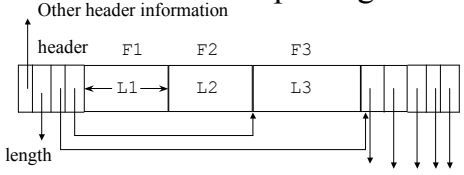
Need the header because:
- The schema may change
  for a while new+old may coexist
- Records from different relations may coexist

Variable Length Records



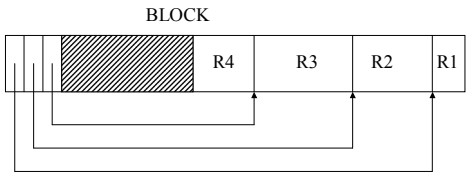
Place the fixed fields first: F1, F2
Then the variable length fields: F3, F4
Null values take 2 bytes only
Sometimes they take 0 bytes (when at the end)

Records With Repeating Fields

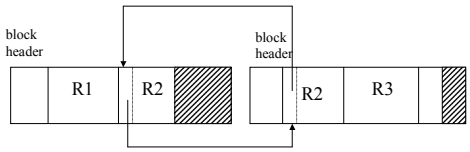


Storing Records in Blocks

- Blocks have fixed size (typically 4k)



Spanning Records Across Blocks



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

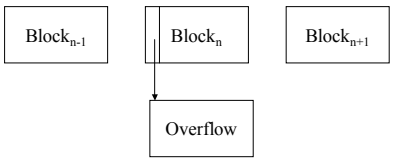
BLOB

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

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



- 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:

Logical address	Physical address
L1	P1
L2	P2
L3	P3

Main Memory Address

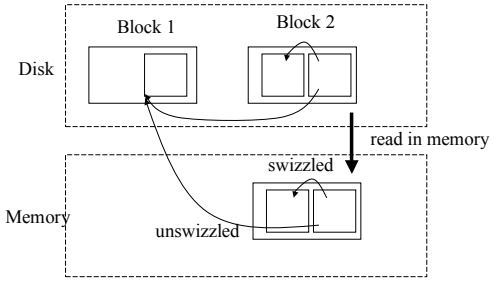
- When the block is read in main memory, it receives a main memory address
- Need another translation table

Memory address	Logical address
M1	L1
M2	L2
M3	L3

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

Pointer Swizzling



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

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