Lecture 19: Representing Data Elements

Wednesday, November 14, 2001

1

Outline

- Representing data elements (3)
- Addresses (3.3)
- Index Structures (4.1, 4.2, 4.3)

2

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

3

Representing Data Elements

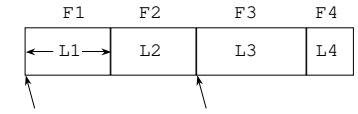
- Representing objects:

```
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)

4

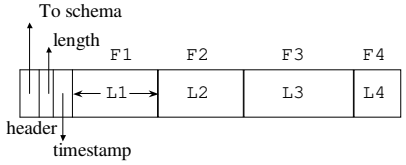
Record Formats: Fixed Length



- 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!**

5

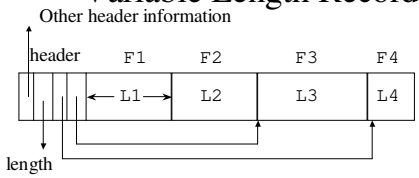
Record Header



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

6

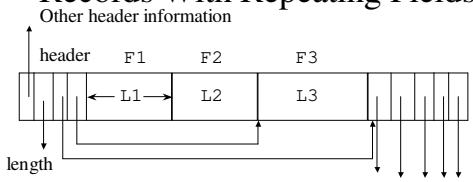
Variable Length Records



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)

7

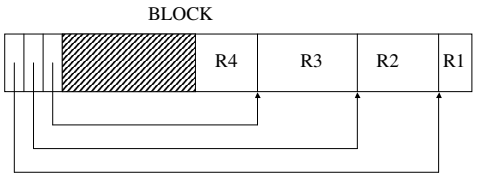
Records With Repeating Fields



8

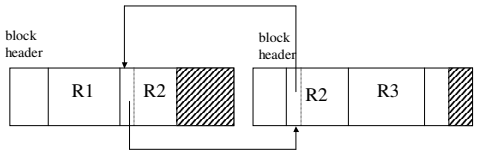
Storing Records in Blocks

- Blocks have fixed size (typically 4k)



9

Spanning Records Across Blocks



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

10

BLOB

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

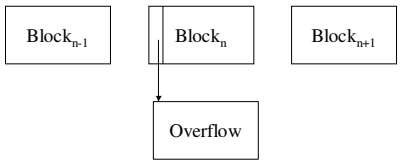
11

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*

12

Overflow Blocks



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

13

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)

14

Modifications: Updates

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

15

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

16

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

17

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

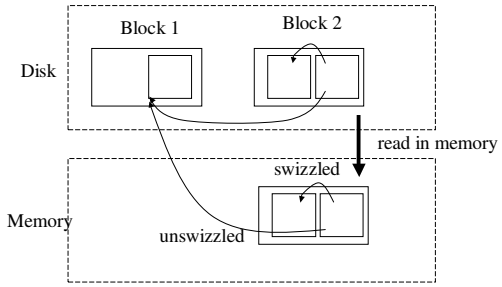
18

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

19

Pointer Swizzling



20

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

21

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

22

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**.

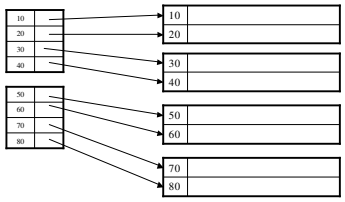
Index Classification

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

24

Primary Index

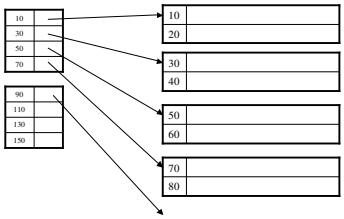
- File is sorted on the index attribute
- *Dense* index: sequence of (key,pointer) pairs



25

Primary Index

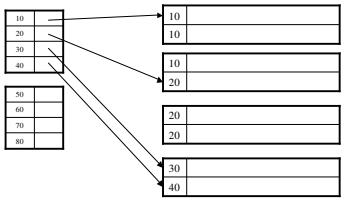
- *Sparse* index



26

Primary Index with Duplicate Keys

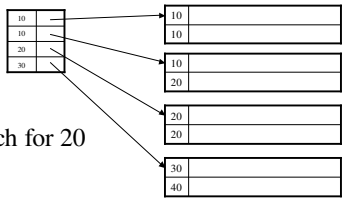
- Dense index:



27

Primary Index with Duplicate Keys

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

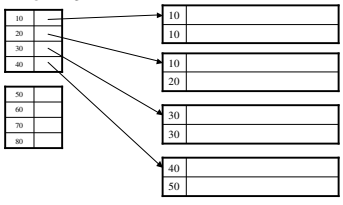


- Search for 20

28

Primary Index with Duplicate Keys

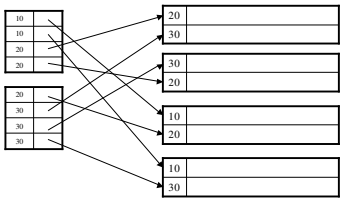
- Better: pointer to lowest new search key in each block:
- Search for 20



29

Secondary Indexes

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



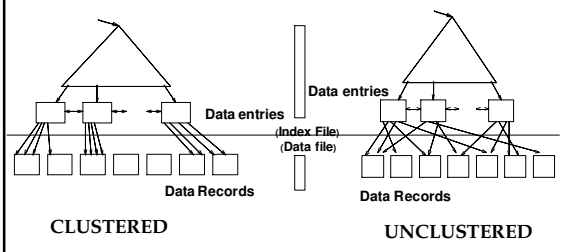
30

Clustered/Unclustered

- Primary indexes = usually clustered
- Secondary indexes = usually unclustered

31

Clustered vs. Unclustered Index



Secondary Indexes

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

33

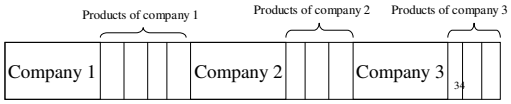
Applications of Secondary Indexes

- *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"



34