Data Storage and Indexing

Dr Paolo Guagliardo

`dbs-lecturer@ed.ac.uk`

THE UNIVERSITY *of* EDINBURGH
informatics

Fall 2018

Files, records and pages

Each table is stored on disk in a file of records

Record: memory area (sequence of bits) logically divided in **fields**

Each record in a file

- corresponds to a row of values in the table
- has the same number of fields
  but **not necessarily the same length**
- has a unique identifier: the record id (**rid**)

Files are organized in pages: blocks of memory of fixed size

The **page size** is a parameter of the DBMS

When data is requested for computation
pages must be fetched from disk and loaded in main memory

File of records

Supports the following operations:

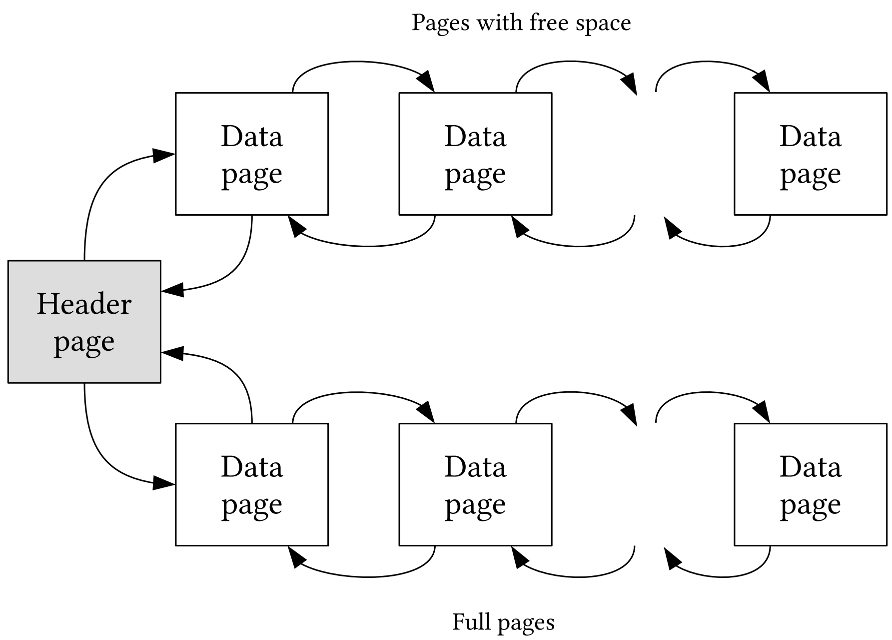
- Insertion of records
- Deletion of records
- Modification of records
- Scan of all records, returned one at a time

Simplest structure: unordered file, called **heap file**

- records are stored in random order across the pages
- supports retrieval of a specific record given its rid

Indexed structures allow to efficiently retrieve records that satisfy a given search condition

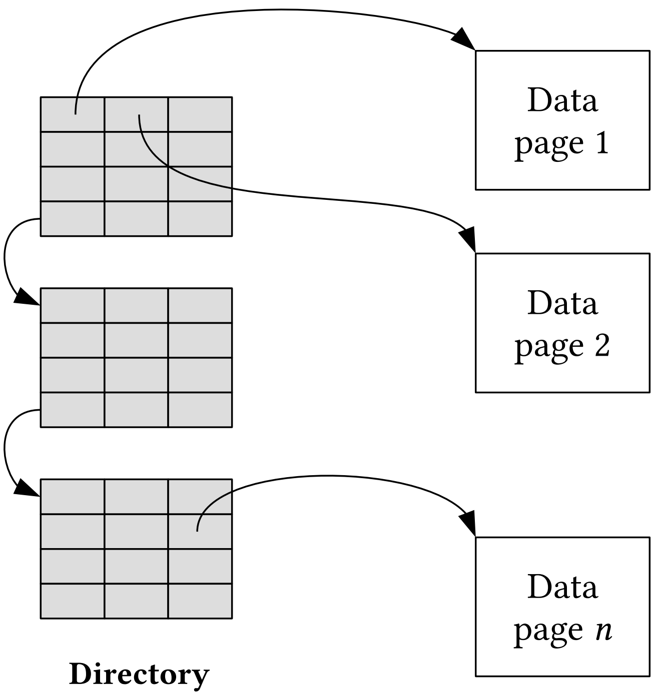
Implementing heap files: Linked list of pages



Disadvantages

- Almost all pages on free list if records are of variable length
- Must scan and examine several pages to insert a record

Implementing heap files: Directory of pages



Free space can be managed by maintaining:

- a bit per entry (free space yes/no)

or

- a count per entry (amount of free space)

Page formats

A page can be thought of as a collection of slots

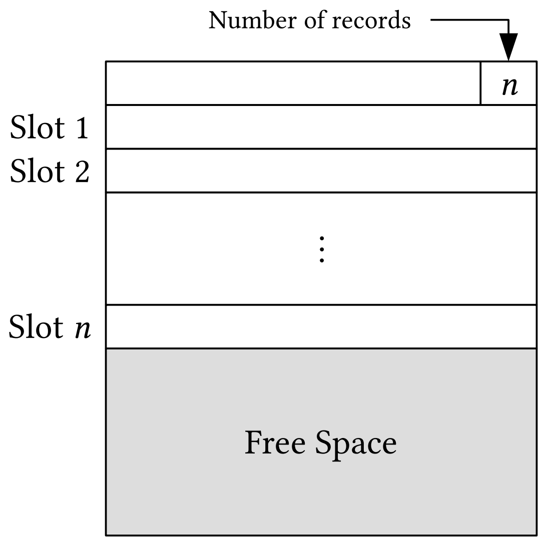
- a record is identified by the **page id** and **slot number** so rid = (page id, slot number)
- alternative: assign unique integer to each record and maintain correspondence between rid and (page, slot)

Format of pages depends on:

- Fixed- vs. variable-length records
- Support for search, insertion, deletion of records

Page formats for fixed-length records

Packed



Records stored in the first n slots

Records located by offset calculation

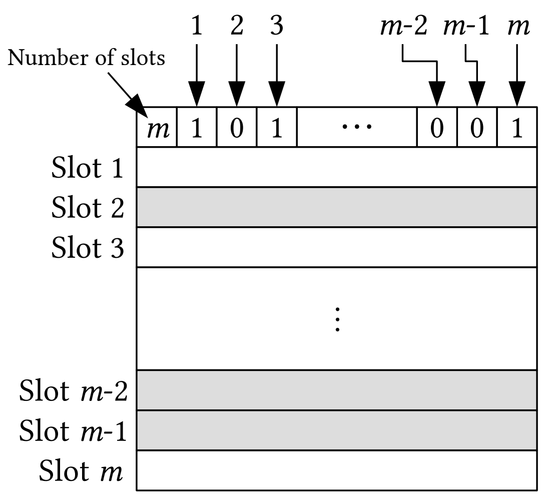
Free space contiguous at the end

When a record is deleted,
the last one is moved to empty slot

Problem if rid contains slot number

Page formats for fixed-length records

Unpacked, Bitmap



Bit array tells which slots are free

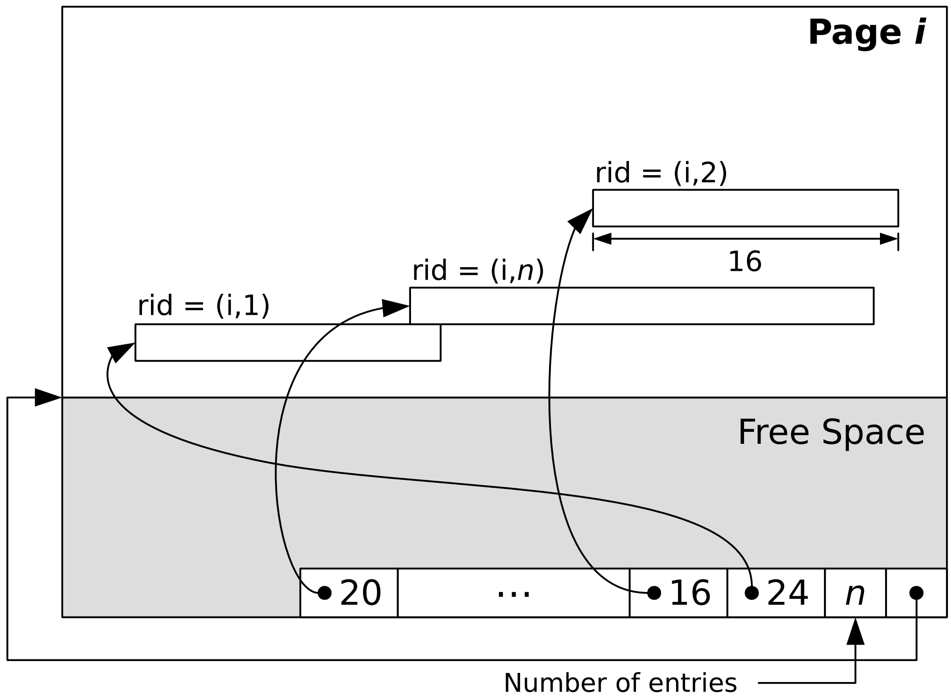
Records located by offset calculation

Scanning all records requires
bit array scan + offset calculation

Insertion of record requires
bit array scan + offset calculation

When a record is deleted,
corresponding bit is turned off

Page format for variable-length records



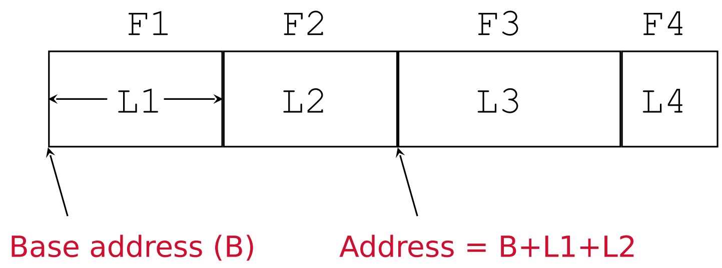
Page format with directory of slots

- Most flexible format
- **Records can be moved without changing rid**
- Can be used also for fixed-length records
- Deletion accomplished by setting slot offset to -1
- Free space must be managed more carefully
  (the page is not pre-formatted into slots)
- Only last slot entry can be removed from directory
- On insertion of record, look for slot entry with offset -1
  (if there is none, add new entry to slot directory)

Fixed-length records

Each field has a **fixed** length (available in the system catalog)

Given the base address B of the record,
the address of field i can be calculated as $B + \sum_{k=1}^{i-1} L_k$

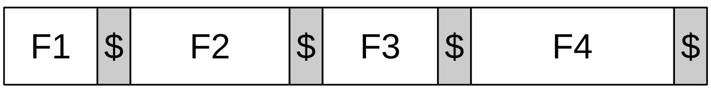


Direct access to fields, but inefficient storage (especially for nulls)

Variable-length records

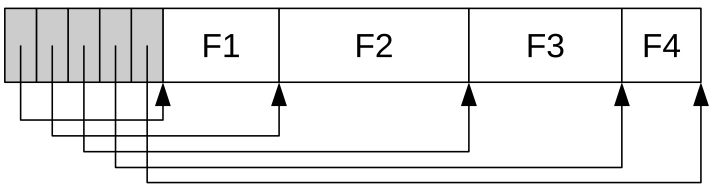
Some of the fields have **variable** length

Fields delimited by special symbol



Access to fields requires a scan of the record

Array of field offsets



Direct access to fields ; efficient storage of nulls

Modifying fields in a record

Potential issues with variable-length records:

- When field grows, shift subsequent fields to make space
- If modified record does not fit in the free space on page, move it to another page leaving a **forwarding address** (we must allocate minimum space for every record)
- If modified record does not fit on any page, split into smaller records across several pages using pointers

Adding fields can cause similar issues in all record formats

Indexing

Index

Data structure that organizes data records based on a **search key** (any subset of the fields of a relation)

- supports efficient retrieval of all data records satisfying a given condition on the search key

Two main indexing strategies

- Hashing (good for conditions based on equality)
- Trees (good for conditions based on ordering)

Hash-based indexing

Organize records into **buckets**
based on a **hash function** h applied to the search key value

For record $\bar{r}$ and search key k

$$h\big(\pi_k(\bar{r})\big) = \text{bucket where } \bar{r} \text{ belongs}$$

Bucket = primary page + zero or more **overflow** pages

Example on blackboard

Tree-based indexing

Records are organised using a **hierarchical tree structure**
that directs the search from the root to relevant pages

Non-leaf nodes contain pointers p separated by search key values v

$$p_0, v_1, p_1, v_2, p_2, \ldots, v_n, p_n$$

For each value v_i

- p_{i-1} points to a node with values less than v_i
- p_i points to a node with values greater than or equal to v_i

Leaf nodes are pages of data records

B-tree
Balanced tree: all paths from root to leaves have same length

Example on blackboard