CS186
Database Systems Notes

Kelvin Lee

UC Berkeley

Contents

1

Disk and Files

1.1 Memory and Disk

- Data must exist in memory in order for a database to access it.
- Once the data becomes large, it becomes impossible to fit all of it within memory.
- **Disks** are used to *cheaply* store all of a database's data, but they incur a large cost whenever data is accessed or new data is written.

1.2 Files, Pages, Records

1.2.1 Records

- A **record** (row) is the basic unit of data for relational databases.
- Records are organized into **relations** (tables) and can be modified, deleted, searched, or created in memory.

1.2.2 Pages

- A **page** is the basic unit of data for disk.
- It is the smallest unit of transfer from disk to memory and vice versa.

1.2.3 Files

- In order to represent relational databases in a format compatible with disk, each relation is stored in its own **file** and its records are organized into pages in the file.
- Based on the relation's schema and access pattern, the database will determine
  1. type of file used
  2. how pages are organized in the file
  3. how records are organized on each page
  4. how each record is formatted.

1.2.4 Choosing File Types

- There are two main files: **heap files** and **sorted files**.
- The databases chooses for each relation which file type to use based on the I/O cost of the relation's access pattern.
- **1 I/O = 1 page read from disk/ 1 page write to disk**.
- I/O calculations are made for each file type based on the insert, delete, and scan operations in its access pattern.
- The file type with less I/O cost is chosen.

1.3 Heap File

A **heap file** is a file type with no particular ordering of pages or of the records on pages and has two main implementations.

1.3.1 Linked List Implementation

- Each data page contains records, a **free space tracker**, and **pointers (byte offsets)** to the next and previous page.
- There is one **header page** that acts as the start of the file and separates the data pages into full pages and free pages.
- When space is needed, empty pages are allocated and appended to the free pages portion of the list.
- When free data pages are full, they are moved from the free space portion to the front of the full pages portion of the linked list. (We move it to the front to avoid traversing the full pages portion to append it)

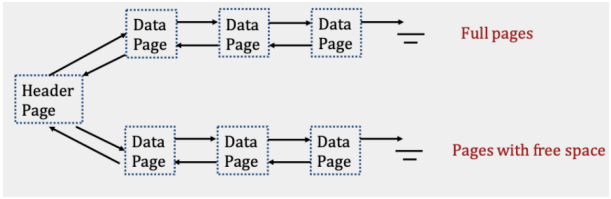


Figure 1.1: Linked List Implementation

1.3.2 Page Directory Implementation

- Differs from the Linked List implementation by only using a linked list for header pages.
- Each header page contains a pointer (byte offset) to the next header page, and its entries contain both **a pointer to a data page** and **the amount of free space remaining within that data page**.

- Since header pages' entries store pointers to each data page, the data pages themselves no longer need to store pointers to neighboring pages.

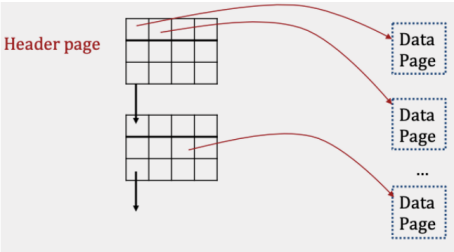
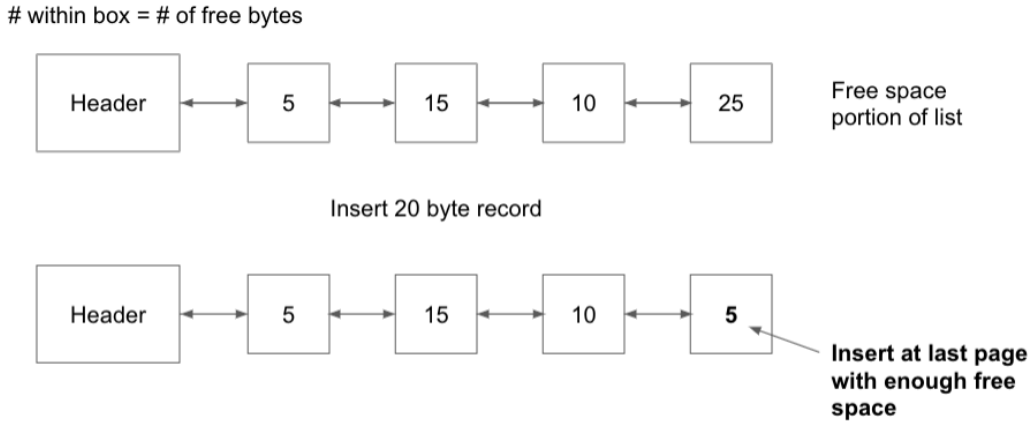


Figure 1.2: Page Directory Implementation

- The main advantage of Page Directories over Linked Lists is that **inserting records is often faster**.
- To find a page with enough space in the Linked List implementation, the header page and each page in the free portion may need to be read.
- In contrast, the Page Directory implementation only requires reading at most all of the header pages, as they contain information about how much space is left on each data page in the file.

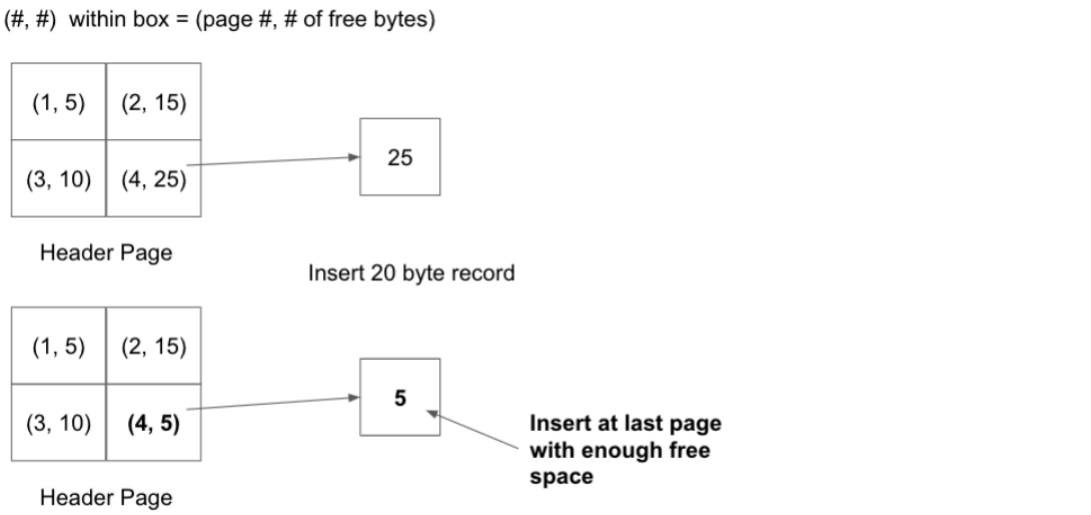
Consider the following example where a heap file is implemented as both a Linked List and a Page Directory. Each page is 30 bytes and a 20 byte record is being inserted into the file:

Linked List



I/O Cost: 5 (read all pages) + 1 (write last data page) = **6 I/Os**

Page Directory



I/O Cost: 1 (read header) + 1 (read data) + 1 (write data) + 1 (write header) = **4 I/Os**

Conclusion:

- Heap files provide faster insertions than sorted files because records can be added to any page with free space, and finding a page with enough free space is often very cheap.
- However, searching for records within heap files requires a full scan every time.
- Every record on every page must be looked at because records are unordered, resulting in a linear cost of N I/Os for every search operation.

1.4 Sorted Files

A **sorted file** is a file type where pages are ordered and records within each page are sorted by key(s).

- These files are implemented using **Page Directories** and enforce an ordering upon data pages based on how records are sorted.
- Searching through sorted files takes $\log N$ I/Os where N is the number of pages since **binary search** can be used to find the page containing the record.
- Insertion, in average case, takes $\log N + N$ I/Os since binary search is need to find the page to write to and that inserted record could potentially cause all later records to be pushed back by one.
- On average, $N/2$ pages will need to be pushed back, and this involves a read and a write IO for each of those pages, resulting in the N I/Os term.

The example below illustrates the worst case. Each data page can store up to 2 records, so inserting a record in Data Page 1, requires a read and a write of all pages that follow, since the rest of the records need to be pushed back.

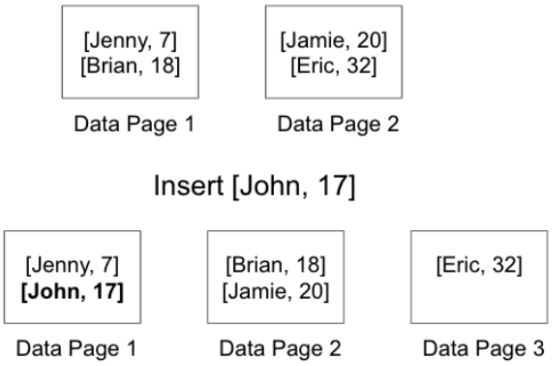


Figure 1.3: Worst case example of inserting record in sorted files