Disk + Files

Memory vs Disk

Whenever a database processes data, it must exist in memory. Accessing this data is relatively fast but once the data becomes very large, it becomes impossible to fit all of it within memory. Disks are used to cheaply store all of a database's data persistently but incur a large cost whenever that data is accessed or new data is written.

Files, Pages, Records

With relational databases, the basic unit of data is a **record** (row). These records are organized into **relations** (tables). We modify, delete, search for, or create records in memory. Whenever we want to persist our changes, we write the data to disk.

With disk, the basic unit of data is a **page** - the smallest unit of transfer from disk to memory and vice versa. In order to represent relations in terms of pages, we organize records into pages and pages into files. Each relation is stored in its own file. Databases use that relation's schema and access pattern to determine: (1) type of file used, (2) how pages are organized in the file, (3) how records are organized on each page, (4) and how records are formatted.

File Formats

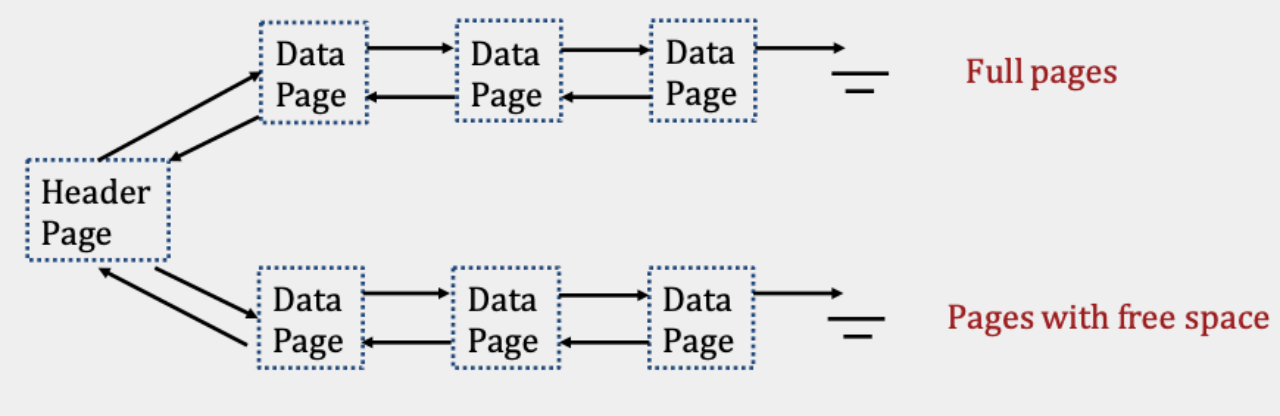
When determining what file format to use, we want to choose the format that results in the lowest number of disk I/Os based on our access pattern (find, scan, insert, etc.). An I/O occurs whenever a page is read from disk or written to disk.

Heap Files

A heap file is a file type with no particular ordering of pages or records on pages.

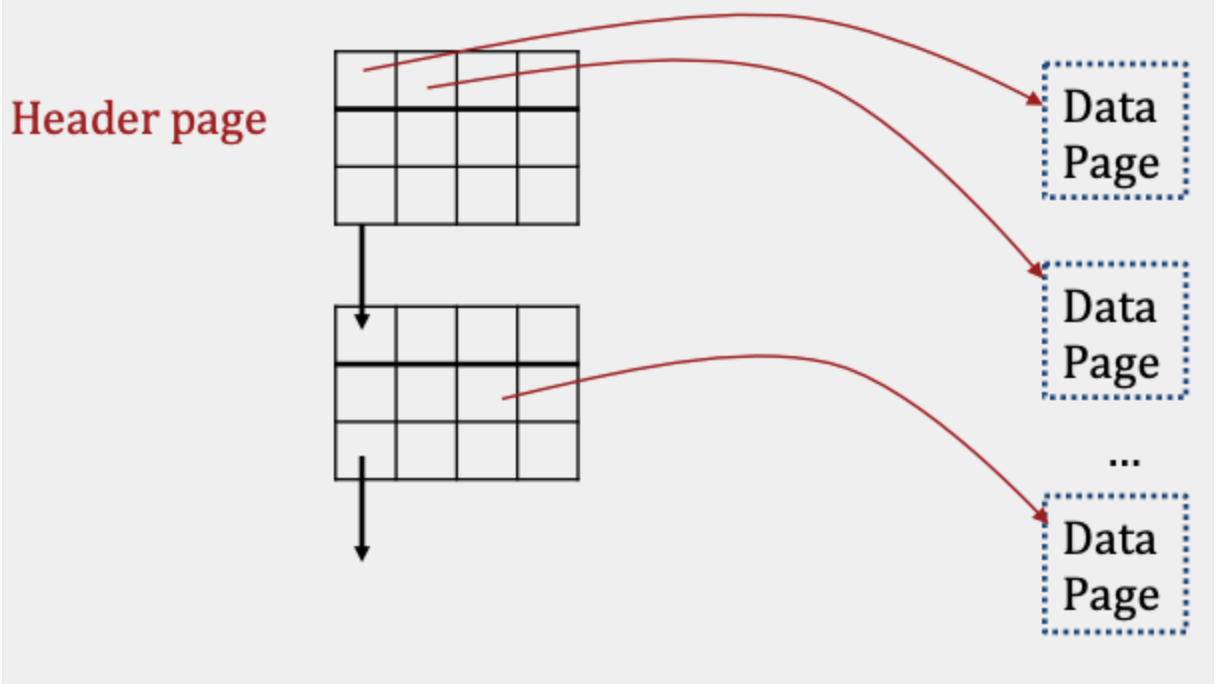
Implementation 1: Linked List

Each page contains **records**, a **free space tracker**, and **pointers** (byte offsets) to the next page and the previous page. There is one header page that serves as the start of the file and is used to separate the data pages into full pages and free pages. Whenever space is needed, free pages are allocated and appended to the list.



Implementation 2: Page Directory

Instead of having a linked list for both header and data pages, the page directory uses a **linked list of just header pages.** Each header page contains pointers (byte offsets) to the next and previous header page along with entries that contain a **pointer to a data page** and information about the **free space within that page**. Since data pages are just used to store records, they no longer track next and previous pointers.



The main advantage of page directory vs linked list is that inserting records is cheaper. Instead of performing I/Os to read each data page and then determining whether there is enough space to insert, only header pages need to be read in order to make that decision.

To highlight this point, assume:

- a heap file is implemented as both a Linked List and a Page Directory
- page size = 30 bytes

You are trying to insert a 20 byte record.

LinkedList (# within boxes represents free bytes remaining)

```
+--------+     +---+     +----+     +----+     +----+
| Header | <-> | 5 | <-> | 10 | <-> | 15 | <-> | 25 |
+--------+     +---+     +----+     +----+     +----+

... INSERT record w/ len 20 bytes ...
```

```
+--------+     +---+     +----+     +----+     +---+
| Header | <-> | 5 | <-> | 10 | <-> | 15 | <-> | 5 |
+--------+     +---+     +----+     +----+     +---+

Cost: 5 I/Os (read all pages) + 1 I/O (write record to last da
ta page) = 6 I/Os
```

Page Directory (tuple within box = (page num, free bytes remaining))

```
                +--------+
                | (1,5)  |                      +------+
Header Page     | (2,10) |        Data Page 4   |  25  |
                | (3,15) |                      +------+
                | (4,25) |
                +--------+

... INSERT record w/ len 20 bytes ...

                +--------+
                | (1,5)  |                     +-----+
Header Page     | (2,10) |       Data Page 4   |  5  |
                | (3,15) |                     +-----+
                | (4,5)  |
                +--------+

Cost: 1 I/O (read header page) + 1 I/O (read last data page) +
1 I/O (write record to last data page) + 1 I/O (write updated
header page) = 4 I/Os
```

This is a simple example and you can imagine that as the # of pages grows, a scenario like this would cause insertion into a heap file implemented as a linked list to be many times more expensive than insertion into a heap file implemented as a page directory.

Heap File Advantages
Insertion is fast as records can be added to any page with free space and finding a free page is cheap when using page directories.

Heap File Disadvantages
Search takes O(N) I/Os where N = # of pages as searching for specific records or a subset of records that match a criteria requires a full scan of the file due to a lack of ordering.

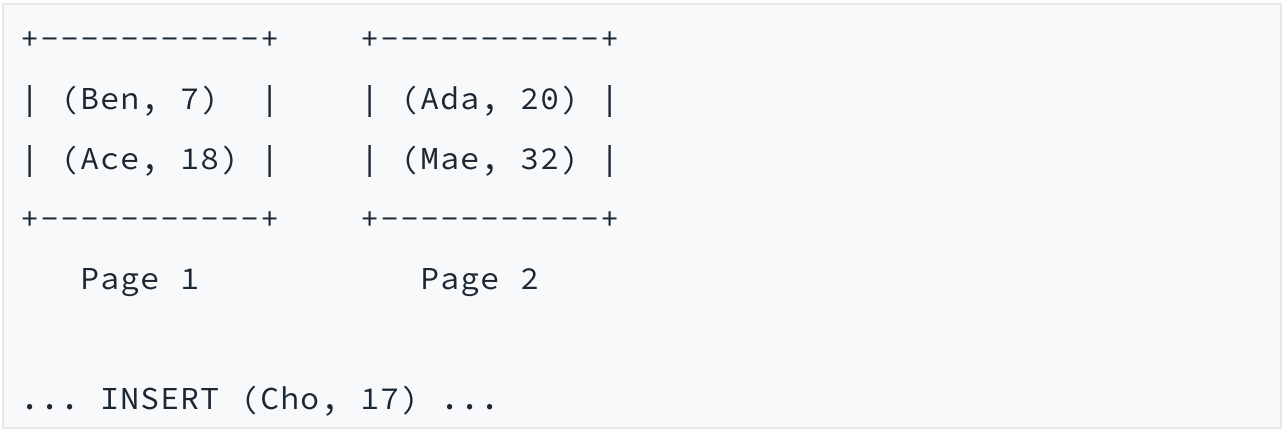
Sorted Files

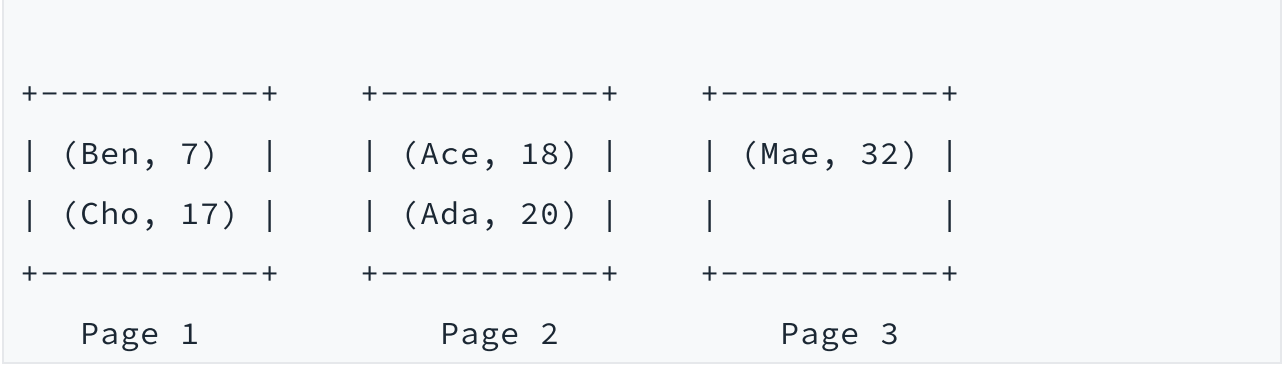
A sorted file is a file type where pages are ordered and records on pages are sorted by key(s).

Implementation: Page Directory
This structure is very similar to the heap file page directory except all data pages must be ordered based on the sorted records they hold.

Sorted File Advantages
Search takes O(logN) I/Os where N = # of pages since binary search can be used to find the page containing the record.

Sorted File Disadvantages
Insertion takes O(logN + N) I/Os as binary search is needed to find the page to write to and that inserted record could potentially cause all later pages to push their records back by one.





Note: heap files with index data structure(s) are typically used over sorted files.

Record Formats

Record formats are determined by the relation's schema and come in 2 types:

Fixed Length Records: records that only contain **fixed length fields** (integer, boolean, date, etc.)
As per their name, FLR all have the same size in terms of bytes.

Variable Length Records : records with both **fixed length** and **variable length fields** (varchar)
VLR store all fixed length fields before variable length fields and use a record header that contains pointers to the end of the variable length fields.

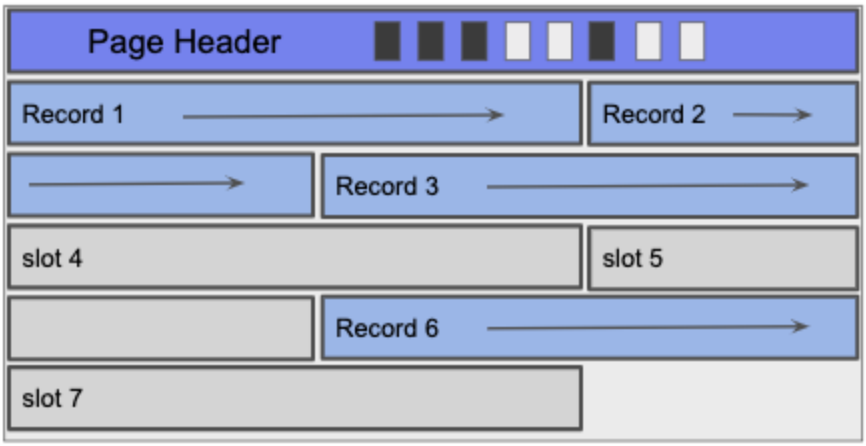
All records can be uniquely identified by a record id: (page #, record #).

Page Formats

Pages with Fixed Length Records
Each page contains a page header that tracks the # of records stored.

If the page is packed, there are no gaps between records. This makes insertion easy as we can calculate the next available position within the page since we know the # of records and the length of each record in the page. Once this value is calculated, we insert the record at that position. Deletion is slightly trickier as it requires moving all records after the deleted record forward by one position to keep the page packed.

If the page is unpacked, the page header typically uses an additional **bitmap** that breaks the page into slots and tracks which slots are open or taken.



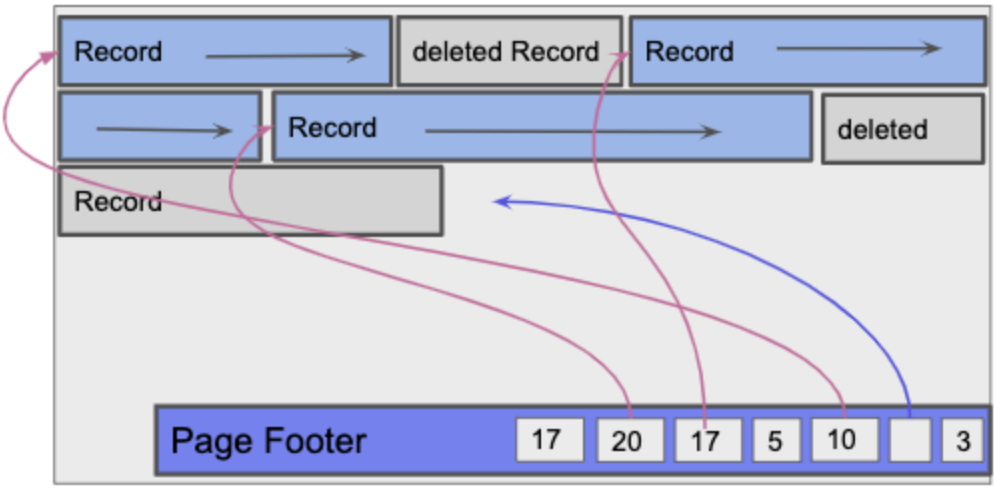
Using the bitmap, insertion involves finding the first open bit, setting the new record in the corresponding slot, and then setting the bit for that slot. With deletion, we clear the deleted record's corresponding bit so that future inserts can overwrite that slot.

Pages with Variable Length Records

The main difference between variable length records and fixed length records is that we no longer have a guarantee on the size of each record. To work around this, each page uses a **footer** that maintains a **slot directory** tracking **record count**, **a free space pointer**, and **entries**. The footer starts from the bottom of the page rather than the top so that the slot directory has room to grow when records are inserted.

The free space pointer points to the next free position within the page and each entry consists of a **record pointer, record length pair**.

If the page is unpacked, deletion involves finding the record's entry within the slot directory and setting the pointer to null. The length is still kept because new records can be inserted using that slot if the new record length is <= old record length. In the case where there are no free entries, the record is inserted and a new entry is added to the slot directory for that record.



If the page is packed, deletion involves removing the record's entry within the slot directory. Additionally, records after the deleted record must be moved forward within the page and their corresponding slot directory entries moved one position back. For insertion, the record is inserted and a new entry is created every time.