CS 186
Fall 2021

Disk and Files

1 Memory and Disk

Whenever a database uses data, that data 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, but they incur a large cost whenever data is accessed or new data is written.

2 Files, Pages, Records

The basic unit of data for relational databases is a **record** (row). These records are organized into **relations** (tables) and can be modified, deleted, searched, or created in memory.

The basic unit of data for disk is a **page**, the smallest unit of transfer from disk to memory and vice versa. 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) and how each record is formatted.

3 Choosing File Types

There are two main types of files: Heap Files and Sorted Files. For each relation, the database chooses which file type to use based on the I/O cost of the relation's access pattern. **1 I/O** is equivalent to **1 page read from disk** or **1 page write to disk**, and I/O calculations are made for each file type based on the insert, delete, and scan operations in its access pattern. The file type that incurs less I/O cost is chosen.

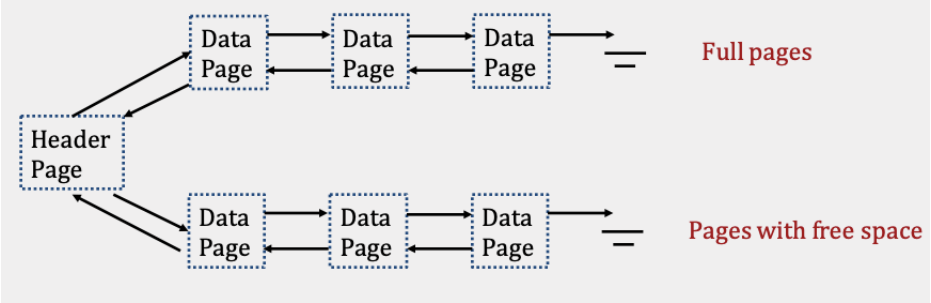
4 Heap File

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

Disk and Files

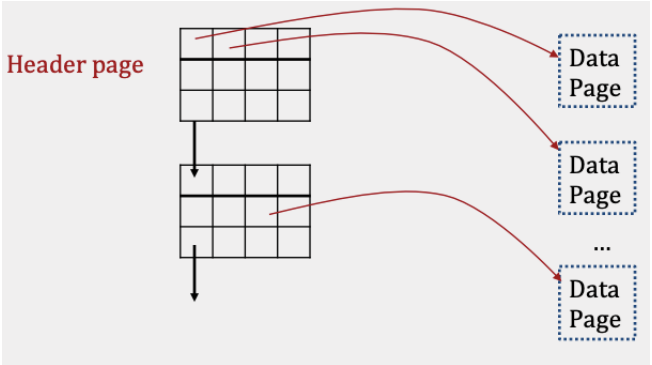
4.1 Linked List Implementation

In the 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 become 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, so we don't have to traverse the entire full pages portion to append it. An alternative is to keep a pointer to the end of this list in the header page. The details of which implementation we use aren't important for this course.



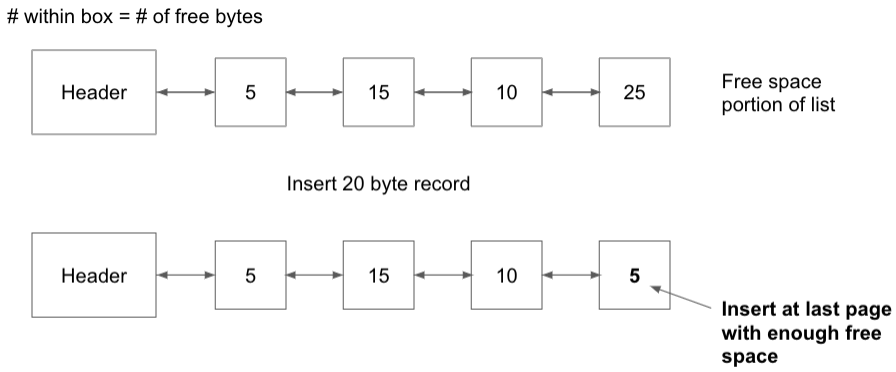
4.2 Page Directory Implementation

The 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 left within that data page**. Since our header pages' entries store pointers to each data page, the data pages themselves no longer need to store pointers to neighboring pages.



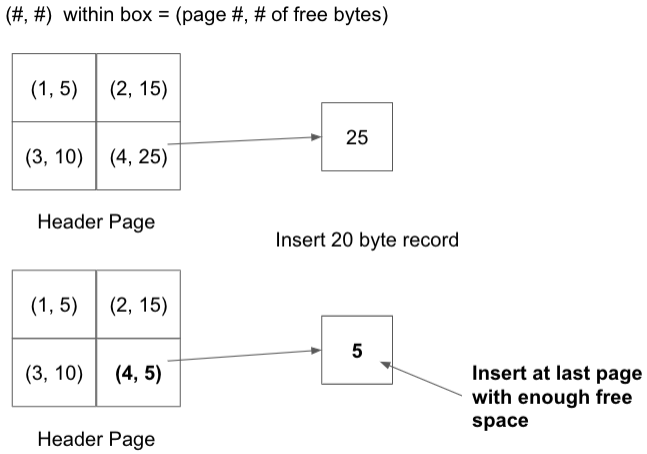
The main advantage of Page Directories over Linked Lists is that inserting records is often faster. To find a page with enough space in the Linked List implementation, the header page and each page in the free portion may need to be read. In contrast, the Page Directory implementation only requires reading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

This is only a small example and as the number of pages increases, a scenario like this would cause insertion in a linked list to be much more expensive than insertion in a page directory.

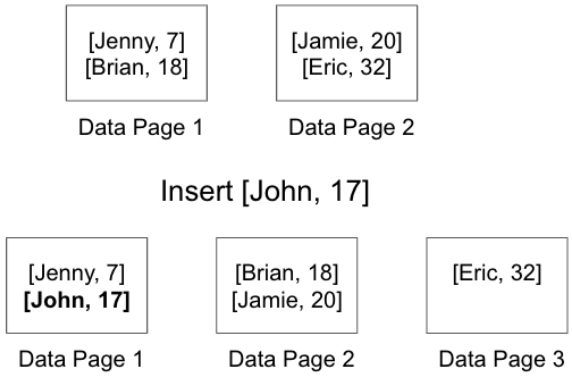
Regardless of the implementation used, heap files provide faster insertions than sorted files (discussed below) because records can be added to any page with free space, and finding a page with enough free space is often very cheap. However, searching for records within heap files requires a full scan 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. We will see that sorted files are much better at searching for recordings.

5 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 logN I/Os where N = # of pages since binary search can be used to find the page containing the record. Meanwhile, insertion, in the average case, takes logN + N I/Os since binary search is needed 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, which results 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.

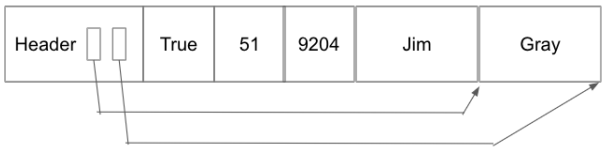


6 A Note on Counting Header Pages

A common area of confusion in counting the cost I/Os of an operation is whether or not to include the cost of accessing the header pages in the file. For all problems in this course, you should ignore the I/O cost associated with reading/ writing the file's header pages when the underlying file implementation is *not* provided in the question. On the other hand, you *must* include the I/O cost associated with reading/ writing the file's header pages when a specific file implementation (i.e., heap file implemented with a linked list or page directory) is provided in the question.

7 Record Types

Record types are completely determined by the relation's schema and come in 2 types: **Fixed Length Records** (FLR) and **Variable Length Records** (VLR). FLRs only contain fixed length fields (integer, boolean, date, etc.), and FLRs with the same schema consist of the same number of bytes. Meanwhile, VLRs contain both fixed length and variable length fields (eg. varchar), resulting in each VLR of the same schema having a potentially different number of bytes. VLRs 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.



Regardless of the format, every record can be uniquely identified by its record id - **[page #, record # on page]**.

8 Page Formats

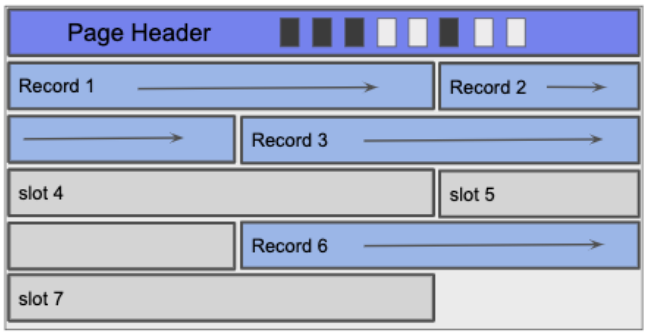
8.1 Pages with Fixed Length Records

Pages containing FLRs always use page headers to store the number of records currently on the page.

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 using the # of existing records and the length of each record. Once this value is calculated, we insert the record at the computed offset. Deletion

is slightly trickier as it requires moving all records after the deleted record towards the top of the page by one position to keep the page packed.

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



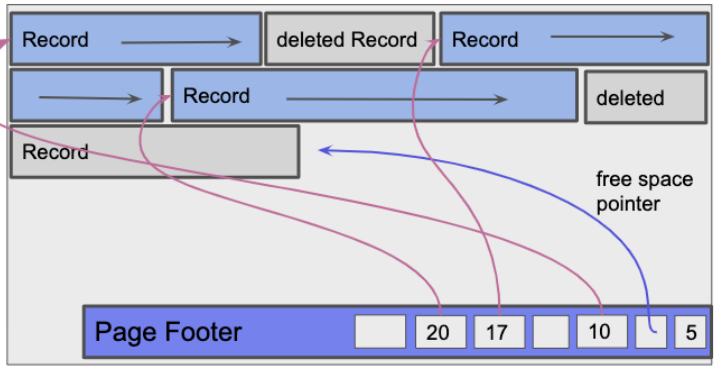
Using the bitmap, insertion involves finding the first open bit, setting the new record in the corresponding slot, and then setting the bit for that slot. With deletion, we clear the deleted record's corresponding bit so that future inserts can overwrite that slot.

8.2 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 **page footer** that maintains a **slot directory** tracking **slot 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 slot count tracks the total number of slots. This includes both filled and empty slots. The free space pointer points to the next free position within the page. Each entry in the slot directory 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 both the record pointer and record length to null.



For future insertions, the record is inserted into the page at the free space pointer and a new [pointer, length] pair is set in any available null entry. In the case where there are no null entries, a new entry is added to the slot directory for that record. The slot count is used to determine which offset the new slot entry should be added at, and then the slot count is incremented. Periodically, records will be reorganized into a packed state where deleted records are removed to make space for future insertions.

If the page is packed, deletion involves setting the record's entry within the slot directory to null. Additionally, records after the deleted record must be moved towards the top of the page by one position and the corresponding slot directory entries shifted towards the bottom of the page by one position. Note that we are only shifting records within the page in which the record was deleted. We are not repacking records across pages for a file. For insertion, the record is inserted at the free space pointer and a new entry is added every time if all slots are full.

9 Practice Questions

1. Given a heap file implemented as a Page Directory, what is the I/O cost to insert a record in the worst case? The directory contains 4 header pages and 3 data pages for each header page. Assume that at least one data page has enough space to fit the record.

2. What is the smallest size, in bytes, of a record from the following schema? Assume that the record header is 5 bytes. (boolean = 1 byte, date = 8 bytes)

   name VARCHAR
   student BOOLEAN
   birthday DATE
   state VARCHAR

3. 4 VLRs are inserted into an empty page. What is the size of the slot directory? (int = 4 bytes) Assume there are initially no slots in the slot directory.

4. Assume you have a heap file implemented in a linked list structure where the header page is connected to two linked lists: a linked list of full pages and a linked list of pages with free space. There are 10 full pages and 5 pages with free space. In the worst case, how many pages would you have to read to see if there is a page with enough space to store some record with a given size?

10 Solutions

1. In the worst case, the only data page with enough free space is on the very last header page. Therefore, the cost is 7 I/Os.

   4 (read header pages) + 1 (read data) + 1 (write data) + 1 (write last header) = 7

2. The smallest size of the VLR is 14 bytes and occurs when both name and state are null.

   5 (record header) + 1 (boolean) + 8 (date) = 14

3. The slot directory contains a slot count, free space pointer, and entries, which are record pointer, record size pairs. Since pointers are just byte offsets within the page, the size of the directory is 40 bytes.

   4 (slot count) + 4 (free space) + (4 (record pointer) + 4 (record size)) * 4 (# records) = 40

4. In the worst case, you have to read the header page and then all 5 pages. Therefore, the cost is 6 I/Os