Discussion 2

Disks, Files, Buffers

Announcements

Project 1 (SQL) is due Thursday (9/10) at 11:59 PM!

Vitamin 2 (Disks, Files, Buffers) is due next **Wed. (9/16) at 11:59 PM**.

Exam Prep Section (SQL & Disks, Files, Buffers) Sunday (9/13) at 5-7 PM

Page/Record Formats

Overview: Files of Pages of Records

- Tables stored as logical files consisting of pages each containing a collection of records

- File (corresponds to a table)
  - Page (many per file)
    - Record (many per page)

Overview: Pages and I/Os

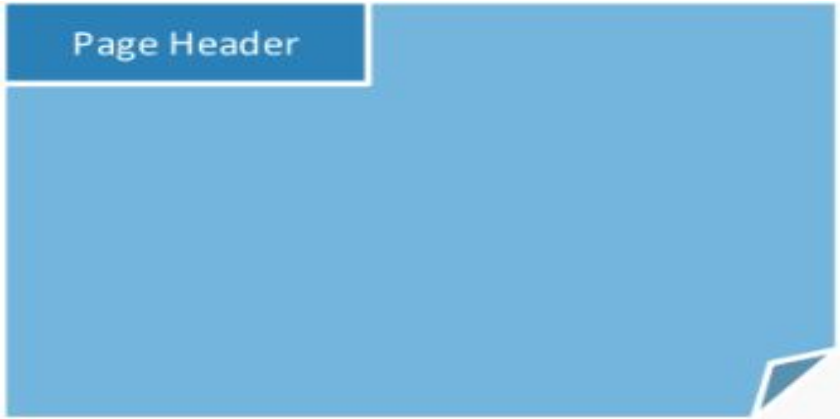
- Pages are managed
  - in memory by the buffer manager: higher levels of database only operate in memory
  - on disk by the disk space manager: reads and writes pages to physical disk/files
- Unit of accesses to physical disk is the page
  - **Cannot** fetch fractions of a page
  - I/O: unit of transferring a page of data between memory and disk (read OR write 1 page = 1 I/O)

Page basics: the header

The **page header** is a portion of each page reserved to keep track of the records in the page.

The page header may contain fields such as:

- Number of records in the page
- Pointer to segment of free space in the page
- Bitmap indicating which parts of the page are in use



Fixed Length Records

Records are made up of multiple **fields** (think: values for columns in a table).

We have **fixed length records** when both the following are true:

- Record lengths are fixed: every record is always the same number of bytes
- Field lengths are consistent: the first field always has N bytes, the second field always has M bytes, etc.

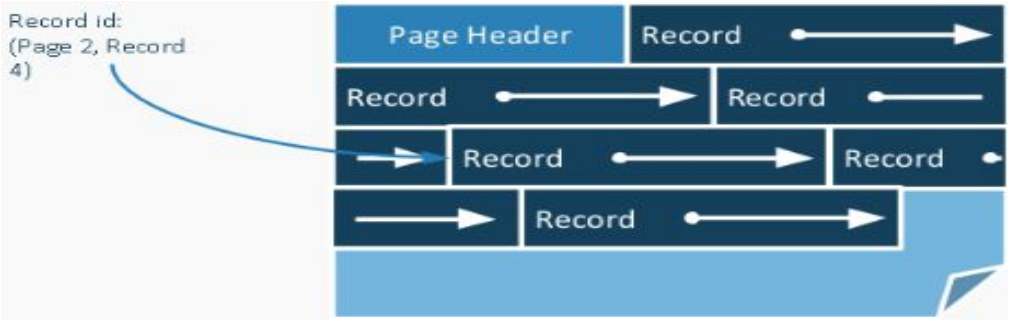
For example:

```
                |  length 5     |  2  |     3     |     4     |
Record1 =       |     Field1    |Field2| Field3   |   Field4  |
Record2 =       |     Field1    |Field2| Field3   |   Field4  |
Record3 =       |     Field1    |Field2| Field3   |   Field4  |
```

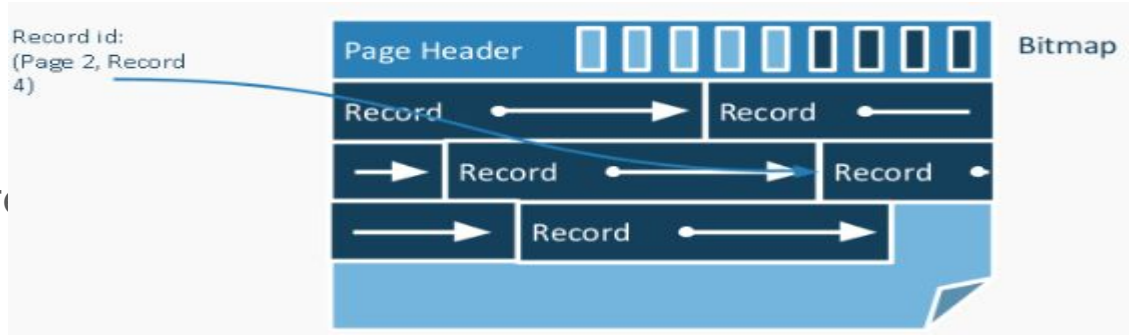
Fixed Length Records

We can store fixed length records in two ways:

- **packed**: no gaps between records, record ID is location in page



- **unpacked**: allow gaps between records, use a bitmap to keep track of where the gaps are



Variable Length Records

We have **variable length records** when we don’t satisfy the conditions for fixed length records, that is, either:

- Record lengths are not fixed: records can take different number of bytes
- Field lengths are not consistent: the third field may take 0 to 4 bytes

Variable Length Records

Two ways to store variable length records:

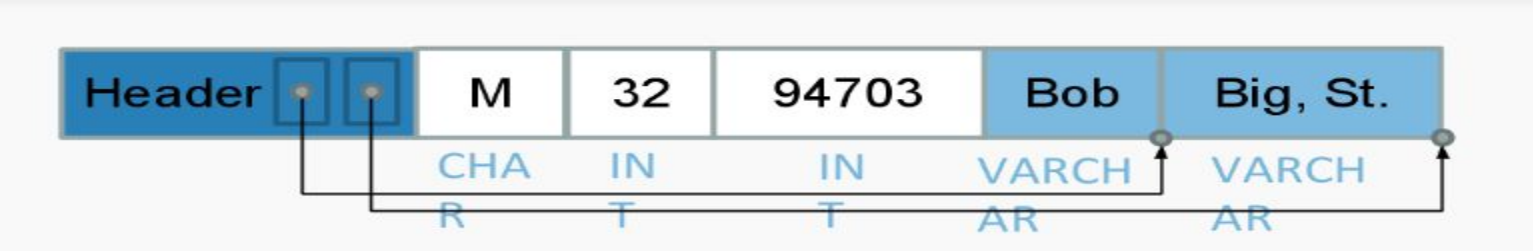
- Delimit fields with a special character ($ in the diagram below)

  - What if F2 contains ‘$’?

Variable Length Records

Two ways to store variable length records:

- Array of field offsets

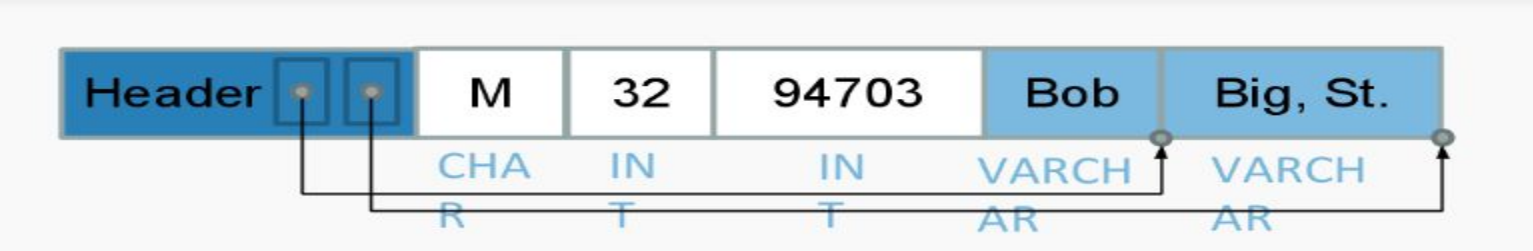


  - Each record contains a **record header**
  - Variable length fields are placed *after* fixed length fields
  - Record header stores **field offset** indicating where each variable length field ends

Variable Length Records

Two ways to store variable length records:

- Array of field offsets



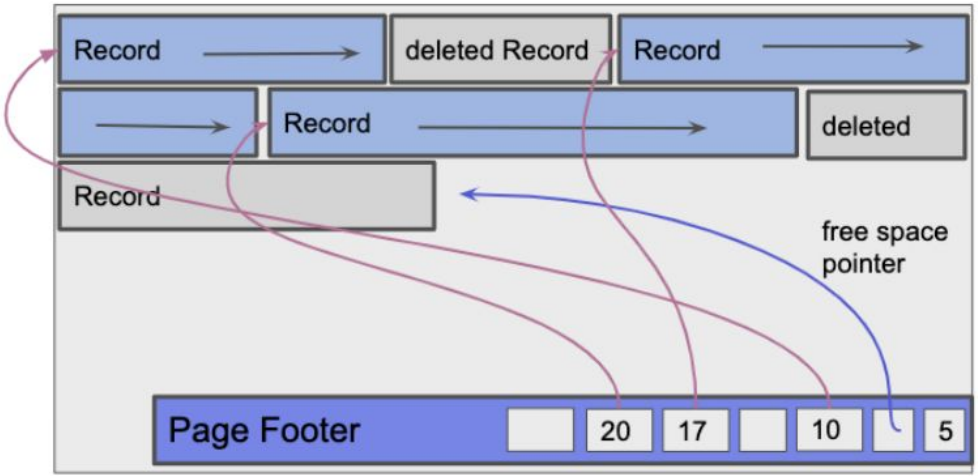
  - An aside: this is not actually sufficient for storing NULLs
    - Cannot distinguish between empty string (“”) and NULL
    - Need some extra metadata (e.g. bitmap in record header or special char in field), which varies widely between different DBMS

Variable Length Records

- How do we know where each record begins?
- What happens when we add and delete records?

Variable Length Records: Slotted Pages

- Move page header to *end* of page (footer) - to allow for header to grow
- Store length and pointer to start of each record in footer
- Store number of slots and pointer to free space



Slotted Page (Unpacked Layout)

Variable Length Records: Fragmentation

- Deleting records causes fragmentation if we use an **unpacked** layout:
  - [3 byte record] [2 byte record] [3 byte record] [2 bytes free space]
  - If we delete the 2 byte record, we have 4 bytes free on the page but cannot insert a 4 byte record

File Organization

Heap Files and Sorted Files

A **heap file** is just a file with no order enforced.

- Within a heap file, we keep track of pages
  - Within a page, keep track of records (and free space)
    - Records placed arbitrarily across pages

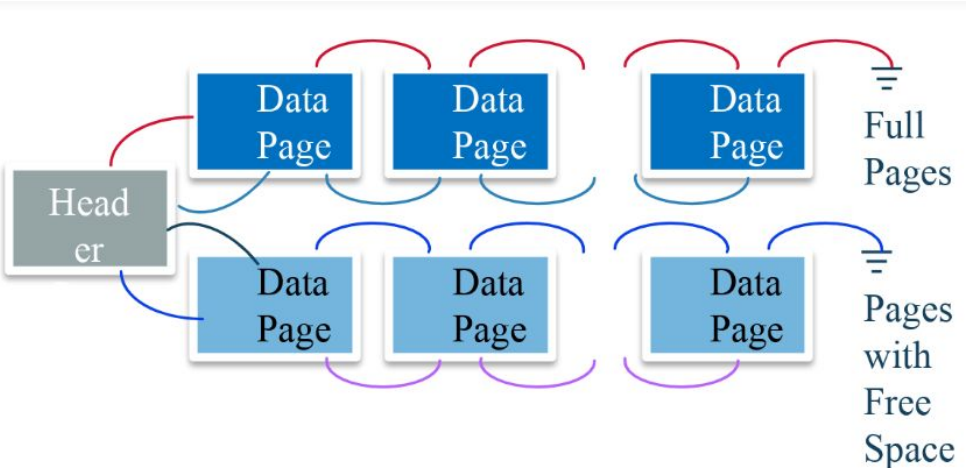
Record ID (RID) = <page id, slot #>

A **sorted file** is similar to a heap file, except we require it be sorted on a key (a subset of the fields).

Implementing Heap Files

One approach to implementing a heap file is as a **list**.

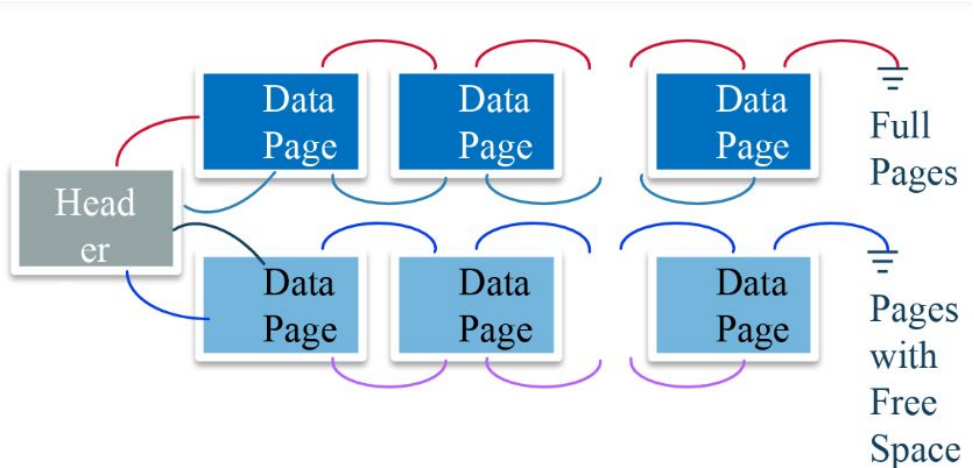
- Each page has two pointers, free space, and data.
- We have two linked lists of pages, both connected to a **header page**
  - List of full pages
  - List of pages with some empty space



Implementing Heap Files

One approach to implementing a heap file is as a **list**.

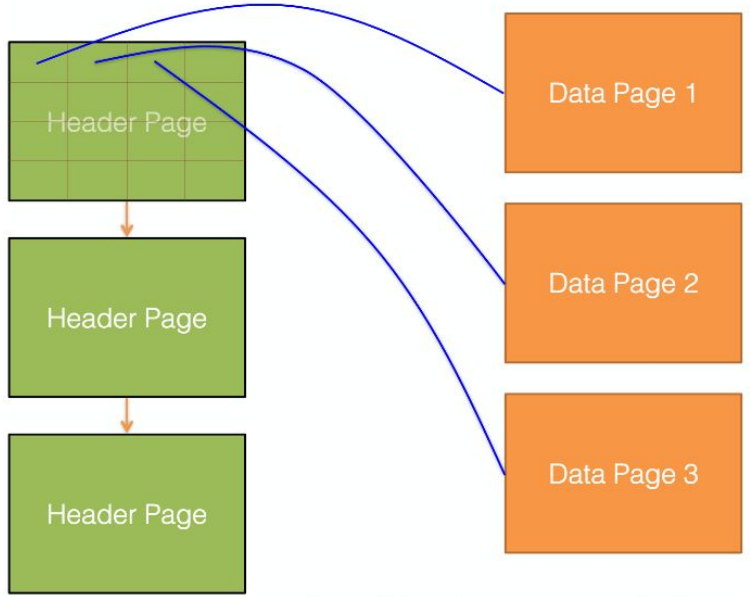
- How do we find a page to insert a 20-byte record in?



Implementing Heap Files

A different approach to implementing a heap file is with a **page directory**.

- We have a linked list of **header pages**, which are each responsible for a set of data pages
  - Stores the amount of free space per data page



Worksheet

True and False - A

When querying for an 16 byte record, exactly 16 bytes of data is read from disk.

True and False - A

When querying for an 16 byte record, exactly 16 bytes of data is read from disk.

False, an entire page of data is read from disk.

True and False - B

Writing to an SSD drive is more costly than reading from an SSD drive.

True and False - B

Writing to an SSD drive is more costly than reading from an SSD drive.

True, a write can involve reorganization to avoid uneven wear and tear (wear leveling).

True and False - C

In a heap file, all pages must be filled to capacity except the last page.

True and False - C

In a heap file, all pages must be filled to capacity except the last page.

False, there is no such requirement.

True and False - D

Assuming integers take 4 bytes and pointers take 4 bytes, a slot directory that is 512 bytes can address 64 records in a page.

True and False - D

Assuming integers take 4 bytes and pointers take 4 bytes, a slot directory that is 512 bytes can address 64 records in a page.

False, we have the free space pointer, which doesn't fit after 64 * (4 + 4) = 512 bytes of per-record data in the slot directory.

True and False - Variable Length Records - B

Variable length records always match or beat space cost when compared to fixed-length record format

True and False - E

In a page containing fixed-length records with no nullable fields, the size of the bitmap never changes.

True and False - E

In a page containing fixed-length records with no nullable fields, the size of the bitmap never changes.

True, the size of the records is fixed, so the number we can fit on a page is fixed.

True and False - Variable Length Records - A

Variable length records do not need a delimiter character to separate fields in the records

True and False - Variable Length Records - A

Variable length records do not need a delimiter character to separate fields in the records

True, using a record header eliminates this requirement.

True and False - Variable Length Records - B

Variable length records always match or beat space cost when compared to fixed-length record format

False, extra space is required for the record header.

True and False - Variable Length Records - C

Variable length records allow access to any field without scanning the entire record

True and False - Variable Length Records - C

Variable length records allow access to any field without scanning the entire record

True, can calculate position of any field using arithmetic.

True and False - Variable Length Records - D

Variable length records have a compact representation of null values

True and False - Variable Length Records - D

Variable length records have a compact representation of null values

True, null values don't take any space (except the pointer in the record header).

Fragmentation and Record Formats #1

Is fragmentation an issue with packed fixed length record page format?

Fragmentation and Record Formats #1

Is fragmentation an issue with packed fixed length record page format?

No, records are compacted upon deletion.

Fragmentation and Record Formats #2

Is fragmentation an issue with variable length records on a slotted page?

Fragmentation and Record Formats #2

Is fragmentation an issue with variable length records on a slotted page?

Yes.

Fragmentation and Record Formats #3

We usually use bitmaps for pages with fixed-length records. Why not just use a slotted page for pages with fixed-length records?

Fragmentation and Record Formats #3

We usually use bitmaps for pages with fixed-length records. Why not just use a slotted page for pages with fixed-length records?

Bitmaps take less space.

Record Formats (a)

Assume we have a table that looks like this:

```
CREATE TABLE Questions (
    qid integer PRIMARY KEY,
    answer integer,
    qtext text,
);
```

Recall that integers and pointers are 4 bytes long. Assume for this question that the record header stores pointers to all of the variable length fields (but that is all that is in the record header). How many bytes will the smallest possible record be?

Record Formats (a)

Assume we have a table that looks like this:

```
CREATE TABLE Questions (
    qid integer PRIMARY KEY,
    answer integer,
    qtext text,
);
```

Recall that integers and pointers are 4 bytes long. Assume for this question that the record header stores pointers to all of the variable length fields (but that is all that is in the record header). How many bytes will the smallest possible record be?

12 bytes. The record header will only contain one pointer to the end of the qtext field so it will only be 4 bytes long. The qid and answer fields are both integers so they are 4 bytes long, and in the smallest case qtext will be 0 bytes. This gives us a total of 12 bytes.

Record Formats (b)

Now assume each field is nullable so we add a bitmap to the beginning of our record header indicating whether or not each field is null. Assume this bitmap is padded so that it takes up a whole number of bytes (i.e. if the bitmap is 10 bits it will take up 2 full bytes). How big is the largest possible record assuming that the qtext is null?

Record Formats (b)

Now assume each field is nullable so we add a bitmap to the beginning of our record header indicating whether or not each field is null. Assume this bitmap is padded so that it takes up a whole number of bytes (i.e. if the bitmap is 10 bits it will take up 2 full bytes). How big is the largest possible record assuming that the qtext is null?

13 bytes. We have 3 fields so there will be 3 slots in our bitmap (and thus 3 bits), meaning our record header will only be 1 byte longer than it was in part a. In the max case, both the qid and answer field will be present and still be 4 bytes each. Therefore, the fields haven't changed at all, and the total record length is now 13 bytes.

Calculate the I/Os #1

Assume we have a heap file A implemented with a linked list and heap file A has 5 full pages and 2 pages with free space, at least one of which has enough space to fit a record.

In the worst case, how many IOs are required to find a page with enough free space?

Calculate the I/Os #1

Assume we have a heap file A implemented with a linked list and heap file A has 5 full pages and 2 pages with free space, at least one of which has enough space to fit a record.

In the worst case, how many IOs are required to find a page with enough free space?

3 I/Os, you read in header and then the 2 pages in the free pages linked list before finding a page with enough free space in the final page.

Calculate the I/Os #2 (Hard!)

Assume we have a heap file A implemented with a linked list and heap file A has 5 full pages and 2 pages with free space, at least one of which has enough space to fit a record.

In the worst case, how many IOs are required to write a record to the 2nd page with free space? Consider what happens when after writing, the page becomes full and assume that the header page can insert at the beginning of the full pages linked list.

Calculate the I/Os #2 (Hard!)

Assume we have a heap file A implemented with a linked list and heap file A has 5 full pages and 2 pages with free space, at least one of which has enough space to fit a record.

In the worst case, how many IOs are required to write a record to the 2nd page with free space? Consider what happens when after writing, the page becomes full and assume that the header page can insert at the beginning of the full pages linked list.

8 I/Os. From the first part, you need 3 I/Os to find the final page. You need 1 I/O to write the record to the final page, 1 I/O to change the pointer from the previous free page, 1 I/O to change the header page (since the final page is now full and in the front of the fullpages), and 2 I/Os to change the previous front of the full pages (read from disk, modify,and write to disk).

Calculate the I/Os #3

Assume we have a heap file A implemented with a page directory. One page in the directory can hold 16 page entries. There are 54 data pages in file A in total.

In the worst case, how many I/Os are required to find a page with free space?

Calculate the I/Os #3

Assume we have a heap file A implemented with a page directory. One page in the directory can hold 16 page entries. There are 54 data pages in file A in total.

In the worst case, how many I/Os are required to find a page with free space?

4 I/Os. There will be ceil(54/16) = 4 pages in the directory. In the worst case, we will need to look through all 4 pages in the directory. (The directory contains the amount of free space for each page.)

Calculate the I/Os #4

Assume we have a heap file A implemented with a page directory. One page in the directory can hold 16 page entries. There are 54 data pages in file A in total.

In the worst case, how many I/Os are required to write a record to a page with free space (assuming at least one free page with enough space to insert a record exists)?

Calculate the I/Os #4

Assume we have a heap file A implemented with a page directory. One page in the directory can hold 16 page entries. There are 54 data pages in file A in total.

In the worst case, how many I/Os are required to write a record to a page with free space (assuming at least one free page with enough space to insert a record exists)?

From #1, we need 4 I/Os to find the free page. We also have to read the page with the free space (1 I/O), write a record to that free page (1 I/O), and write to the page directory (1 I/O), for a total of 7 I/Os.