Heap file system

Dr. Yihsiang Liow (March 10, 2020)

Contents

Chapter 1

Heap file system

A heap file system is a container of records. The system is divided into layers:

- At the highest level, the heap file system is a container of pages where each page is sequence of bytes. The number of bytes is a page is related to the size of a read/write block of bytes on a storage device. A storage block size of 512 bytes is common. A page in memory is therefore either a block or a multiple of blocks. (It doesn't make sense for a page in memory to correspond to 1/2 a block.) For instance it might be 16KB. I only need to think in terms of page size. The file I/O subsystem will take care of the actual block read/write operations. But in general, for a real RDBMS, the page size needs to be chosen carefully since if the page size is too big, it will take a long time for the file I/O subsystem to complete the read/write operation to transfer all the blocks in the page.
- At the next (lower) level, a page is a container of records.
- This is then followed by a record which is a container of fields.

Dividing the system into the above layers is helpful in understanding the system and is also helpful in designing the software for such a system.

Note that at this point, I'm not going to focus on fast search. I'll talk about speeding up the search when I talk about index structures.

Of course since the heap file system is a storage engine for records, we want the following operations on the heap file system:

- insert a record
- delete a record
- find a record

Since I'm not going to talk about fast search through an index structure, the find operation just need a way to iterate over the records of the heap file system.

1.1 Page: a container of records

In this section "pointer" means byte offset from the beginning of a page. So a pointer is 0 means it's pointing to the first byte of the page.

In this section, I'll talk about how a record is stored in a page. Here, a record is an array of bytes. I going to assume that every record I am interested in can be stored in a page. (Later, I'll consider the case where a record spills across several pages.)

1.1.1 Doubly-linked list version 1.

If a heap file (or table) can be stored in a single page, then here's one option:

header		
free		
prev	next	record
prev	next	record
prev	next	record
prev	next	record

The table header contains the head and tail of the record-with-next-and-previous pointer and pointer and length of the free space. Here a pointer is a byte offset value. The head, tail, next, previous pointer in this case are byte offset within this page. We may assume that all pointer values are 16-bit wide. The actual number of bytes (or bits) for a pointer depends on the designer. In this case, the records are organized as a doubly-linked list.

Note that I'm not assuming the that records have the same length.

There might be other useful information in the header. For instance one can store the number of records in the header.

1.1.2 Doubly-linked list version 2.

Of course when a record is deleted, we don't really want to move records around so that unused space is contiguous – this is costly. So besides the contiguous free space, we also maintain a doubly-linked list of free nodes. The head and tail of the free list is also stored in the header.

1.1.3 Doubly-linked list version 3.

Another variation is where the (prev, next) values are in the section of the page as (prev, next, record pointer):

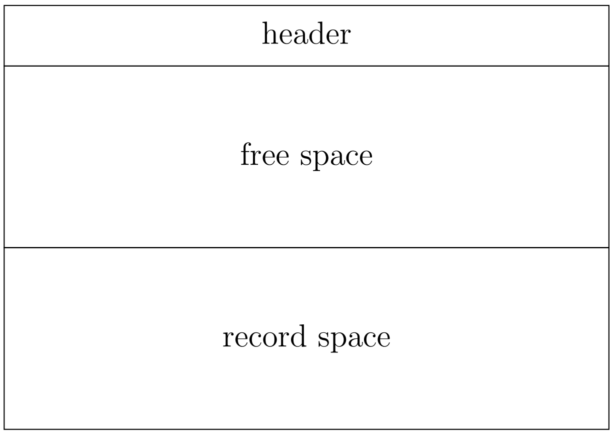
header
linked list nodes of (prev, next, record pointer)
free
record
record
record
record

In this case, the next and prev points to the offset of the (prev, next, record pointer) in the header.

1.1.4 Array version 1.

Another possibility is to keep an array of pointers in the header. Specifically, the page is divided into three sections: header, free space, records space.

1. header: This contains
   - (pointer, length) of free space
   - number of (pointer, length) of records
   - array of (pointer, length)s
2. free space: The first item in the header points to this space.
3. record space: The array of pointers in the third section of the header points to records in this area.



For instance if the header contains

$$(26, 200), 5, (340, 6), (330, 5), (320, 9), (310, 7), (300, 5)$$

then it means that

1. the free space starts at byte 26 and has a length of 200
2. there are 5 records:
   a) the first record starts at byte 340 and (uses 6 bytes).
   b) the second record starts at byte 330 (and uses 5 bytes),
   c) the third record starts at byte 320 (and uses 9 bytes),
   d) the fourth record starts at byte 310 (and uses 7 bytes),
   e) the fifth record starts at byte 300 (and uses 5 bytes),

The array of (pointer, length)s in the header is sometimes called the **slot directory**.

If a record is to be added, the memory used is at the *end* of the free space. An entry is added to the slot directory and the number of records in the header is updated. Notice that the header grows *down* while the record space grows *up*.

If a record is deleted, the page offset of the associated slot entryis set to -1. For instance if the record at byte 330 is deleted, then the header becomes

$$(26, 200), 5, (340, 6), (-1, 5), (320, 9), (310, 7), (300, 5)$$

The size of the integers in the header depends on the implementation. For instance 16-bit integers would give you values in the range -32768..32767. So if a page is 32KB (i.e., 32768 bytes), then the offset value is 0..32767. Therefore if your page size is 32KB, then you can use 16-bit integer for page offset and length values.

Note that the data in the slot directory for a deleted record is not removed. Why? A record can be identified by

(page number, index in the slot directory)

This tuple is called the **record id** (**rid**). Note that the rid identifies a record in the *heap file system* and not in a page. Obviously if you need to access a record by its rid, that means that the associated (pointer, length) data in the slot directory cannot be deleted and cannot be moved in the slot directory.

1.1.5 Array version 2: directory as header.

Now note that the offset for the free space in fact can be computed. Therefore we do not need to include the offset for the free space. The header from version 1

$$(26, 200), 5, (300, 5), (310, 7), (320, 9), (330, 5), (340, 6)$$

becomes

$$200, 5, (300, 5), (310, 7), (320, 9), (330, 5), (340, 6)$$

The byte offset of the free space f is $2 + 2 + 5 \times 4 = 24$ bytes away. If 2 bytes are used for the integers in the header, then in general the free space offset f

$$f = 4(r + 1)$$

where r is the number of records. If the free space length, l, is the number of bytes in the free space, then the byte just after the free space is

$$f + l$$

If a new record of size n is to be added and it is saved in the free space (and we need 4 bytes for the slot entry), then the byte offset for the record is

$$f + l - n = 4(r + 1) + l - n$$

since we are using the end of the free space for allocation of new records. And of course we have to make sure that it does not cross over into the header, i.e.,

$$n + 4 \leq l$$

Of course if f is stored, then the number of computations will be less (however you need to stored and updated f). In summary:

1. Enough space for new record:
$$n + 4 \leq l$$
2. Offset of new record:
$$4(r + 1) + l - n$$

1.1.6 Array version 3: directory as footer.

In the above, we have to compute the offset of the record to be allocated by

$$4(r + 1) + l - n$$

since we are using the end of the free space. What if we use the beginning of the free space? Then the offset is just
$$4(r + 1)$$
which seems to be less computations. However if we do this, then we would have to constantly move the record space when the header increases in size. This is a very bad idea.

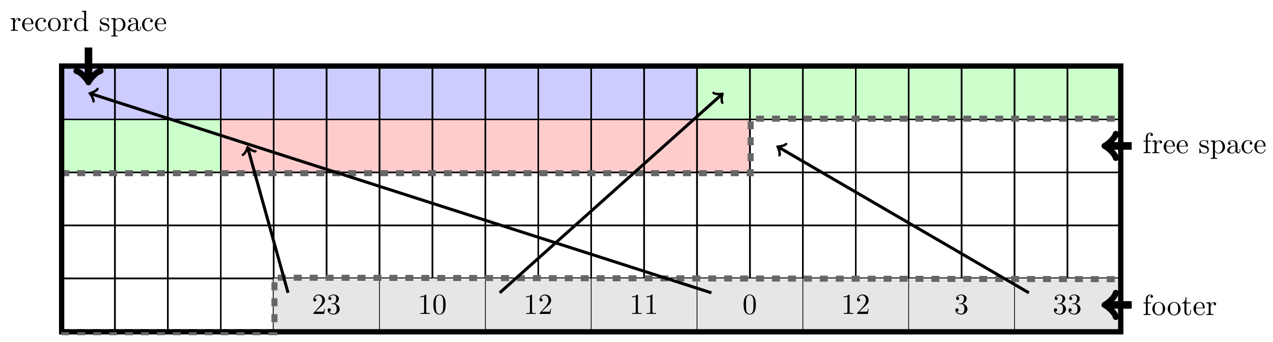
Another option is to do this:

1. record space
2. free space
3. footer (contains offset of free space and slot array)

The array of (offset, length) values are stored in the opposite order, where index 0 nearer to the end.

The following picture shows you a page with the following information:

1. Each offset and length takes up 2 bytes
2. The record space is from offset 0 to 32
3. The free space is from offset 33 to 83
4. There are 3 records in the record space
   a) The first record is at offset 0 and has length 12.
   b) The second record is at offset 12 and has length 11.
   c) The third record is at offset 23 and has length 10.



Note that if there are r records, then the number of bytes used by the footer is

$$2 + 2 + 4r = 4(1 + r)$$

If a new record of size n is to be added to the page with free offset f, then, together with 4 bytes for the new value in the slot array, there's enough space for the new record if

$$f + 4(1 + r) + n + 4 \leq p$$

i.e.,

$$f + 4(2 + r) + n \leq p$$

where p is the page size. The offset for this new record is at

$$f$$

In summary:

1. Enough space for new record:

$$f + 4(2 + r) + n \leq p$$

2. Offset of new record:

$$f$$

When a record is deleted, the page offset value in the corresponding (page offset, length) value in the array is set to -1 – it is not removed. If a record is modified, as much as possible, the record value stays in the original location. If it is impossible to fit in at that location, the record is moved to the free space.

[Note. Because the footer is at the end of the page and the free offset pointer is in fact the last, this means that when this page is first fetched, the buffer manager must wait for the whole page to load before an insert is possible. In a highly parallelized system where the RDBMS reads and writes to partially fetched page, the version that uses a header rather than footer is better. Another thing to note is that if we use a header instead of footer, the offset of 0 cannot be used as a pointer to record. Therefore is a record is deleted, instead of using -1 for the pointer, we can use 0 – therefore the pointer can be an unsigned int.]

1.1.7 Array version 4.

Yet another possibility is to keep an array of (page offset, length, not-deleted flag) in the header. For instance if the header contains

$$200, 4, (300, 5, 1), (310, 7, 1), (320, 9, 1), (330, 5, 0), (340, 6, 1)$$

there are 4 records starting at 300, 310, 320, 340 and there's a deleted record at 330. Because the pointer of a deleted record is preserved, in this case, the "holes" in the record space can possibly be reused. In the case of Array version 2 and 3, the pointer is set to -1 which means that these "holes" cannot be located quickly. However, note that RDBMS needs to run as fast as possible. Usually trying to save space by reusing deleted records (instead of the free space) is not the best strategy for speed.

Our RDBMS. For our RDBMS, we will use Array Version 3.

1.2 Record: a container of fields

```
http://www-db.deis.unibo.it/courses/TBD/Lezioni/01%20-%20PhysycalDBMS.pdf
```

A record is made up of two parts:

- the fields (i.e., the data)
- a header that describes how to read the fields (i.e., the metadata)

First I'll talk about the fields.

1.2.1 Version 1.

Here is the structure of a record of fields:

null bits	fixed length fields	offset-length pairs	variable length fields

The number of bytes used by the null bits depend on the number of fields. For instance if there are 4 fields, then the null bits field uses 1 byte. If there are 10 fields, then the null bits field uses 2 bytes. If a bit in the null bits is 1, it means that the associated field has a value. If a bit in the null bits is 0, it means that the associated field does not hold a value.

Assuming the above is stored in memory at address `p`, then the address of any field can be determined by the following information:

1. The number of fixed width fields
2. The widths (in bytes) of the fixed width fields
3. The number of variable width fields

The following can also be easily computed:

4. The number of fields
5. The number of bytes used by the null bits field
6. The starting address of the offset-length pairs

For instance suppose I want to store 42 (as a fixed width 32-bit integer), 123 (as a fixed width 32-bit integer), `"hello"` as a variable length field of width 15 bytes, and `"world!"` as a variable length field of width 10 bytes. The record (in memory or storage) then looks like this:

0	1	5	9	13	17	32
00001111	42	123	(17,15)	(32,10)	hello	world!

The numbers on top are the byte offset from the beginning of the record. The the null bits is `00001110`, then the first field does not hold a value (i.e.., it is NULL).

Note that each offset-length pair occupies 4 bytes, 2 bytes for the offset and 2 bytes for the length – both are 16-bit unsigned integer.

Notes.

1. Instead of thinking of fixed width and variable width fields, think of a fixed width field as a field where the width information is not stored in the record. Think of a variable field as a field where the starting byte offset location and width is stored within the record. (Note that there's some redundancy – the offset of the first variable width field need not be stored in the record.)
2. For the variable length fields, the fact that a field is NULL can be represented by the fact that the length is 0. However, fixed length fields do not have lengths. So you still need the null bits for fixed length fields. Note that for the case of a variable length field, there's an (offset, length) value and the value at that location has a representation for the variable length data that might contain another width value. The two width values are (clearly) not the same.

1.2.2 Version 2

A variation of the above is to have the offsets (i.e. pointers) as an array at the beginning of the record. The size of the this array (i.e., number of offsets) is the same for every record of the relation. Therefore this should not be saved in the table, but as a metadata somewhere else.

Using this version, the above

0	1	5	9	13	17	32
00001111	42	123	(17,15)	(32,10)	hello	world!

becomes

0	2	4	8	8	10	14	18	33
10	14	18	33	43	42	123	hello	world!

assuming each offset in the offset array occupies 2 bytes. Note that there is one extra pointer that points to the byte just after the last field. Therefore there are 5 points for 4 fields. Why? Then the difference between consecutive pointers will give you the length of each field. And the fact that the difference is 0, tells you that field is NULL.

Our RDBMS. For our RDBMS, we will use Version 1.

Question. Compare the above two record layout. What are the pros and cons?

Notes.

- There are usually other information besides the field-related data. For instance you usually include datetime stamp for the creation, access, and modification of the record. This would be useful for transaction processing.
- Note the record structure treats each field as a array of bytes. When applying this file system to RDBMS, each field has an attribute name, data type, etc. Such information (the schema for this record) is not stored in the record since records from the same relation would contain the same schema information. Therefore schema information is stored separately. The storage of such information is called the system catalogs.

1.2.3 Record header

Now let's talk about the header of the record.

Note that the above (version 1, 2) talks about the fields in a record. A record itself has a header which will be used to read the fields in the record. The header usually includes the following:

- Miscellaneous information such as a length of the header and record.
- Some form of ID for the schema of the record. The schema would contain information on the number of fixed width fields, number of variable width fields, the name of the fields, the data type of the fields, etc.

For transaction processing there might also be information such as datetime stamp of when the record was created, when it as last accessed, and when it was last modified; there might also be lock information on the record.

1.3 Heap file system: container of pages

What happens if the records of a relation (i.e. table) cannot fit into a page? (For instance in the case where you want to store a very long string or even a video file.) Also, there are usually many relations.

Instead of mapping a relation directly to a page that stores records:

$$\text{table name} \rightarrow \text{page number}$$

we of course need to map a relation to a collection of pages since a relation might have more rows that can fit onto a single page:

$$\text{table name} \rightarrow \text{page numbers}$$

This assumes that your storage holds multiple relations.

Another implementation option is to have one heapfile system storing records from only one relation.

1.3.1 Version 1

One way to organize the pages is to link up the pages as a doubly-linked list: for instance at the beginning of the page, we can keep two extra integers, one for the previous page and one for the next. (Of course the page number of the head and tail pages has to be kept somewhere.)

1.3.2 Version 2

Another way is to map a relation to a doubly linked list of **directory pages**. A directory page is just a page containing page numbers.

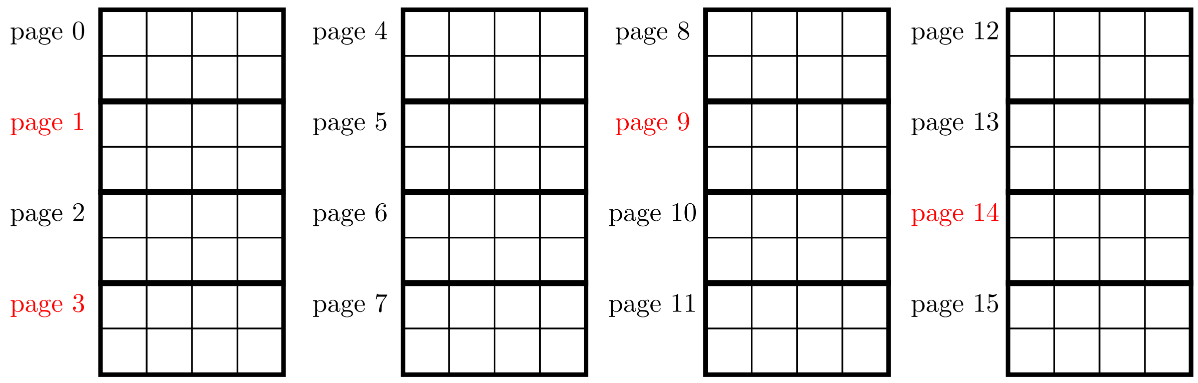
$$\text{table name} \rightarrow \text{directory pages} \rightarrow \text{page numbers}$$

Directory pages for a relation form a doubly linked list. Information in the directory pages:

1. previous directory page number
2. next directory page number
3. number of record pages that this directory page points to
4. array of (page number of records page, size of free space in this records page) in this page

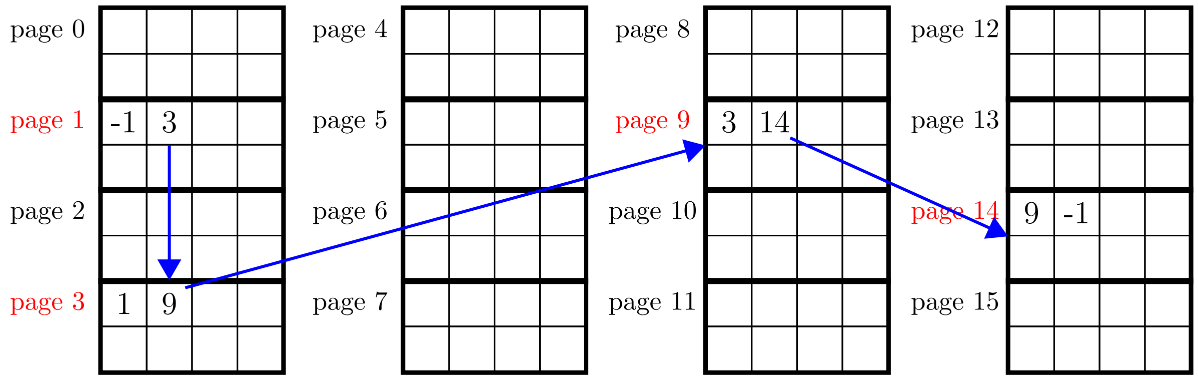
(Why doubly linked and not singly? Because if you have deleted all records pages for a directory page, then you need to put this directory page into the free list.)

Here's an example. Suppose for relation Persons, the directory pages are 1, 3, 9, 14:

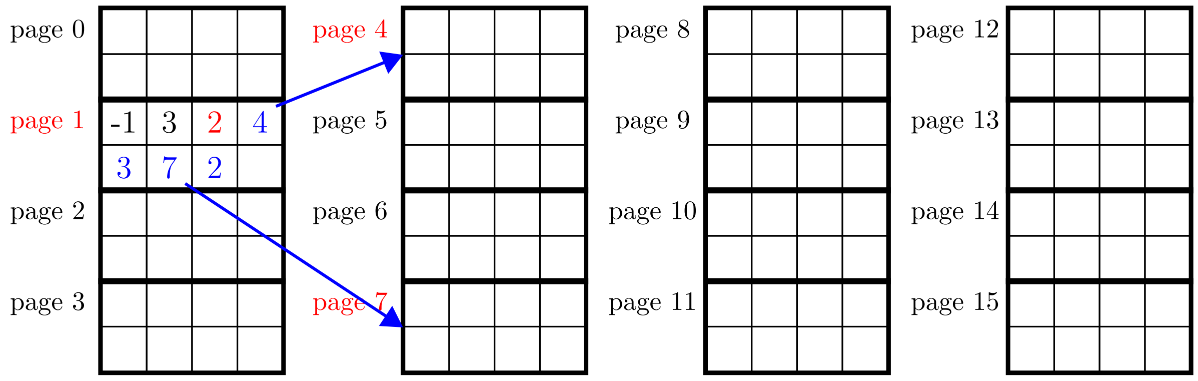


(This is just for illustration. Of course real pages should be much larger.)

These pages form a doubly linked list (the first number in a page is the previous page number in the linked list and the second is the next page number):



The directory page at page 1 points to records pages:



In this case, page 1 points to 2 records pages (page 4 and 7):

1. The first records page is page 4 which has 3 free bytes.
2. The second records page is page 7 which has 2 free bytes.

Note that at this point a page is either a records page or a directory page.

In the above, you notice the following organizational principle:

1. Each records page contains records from one relation.
2. Each directory page contains page numbers pages of one relation.

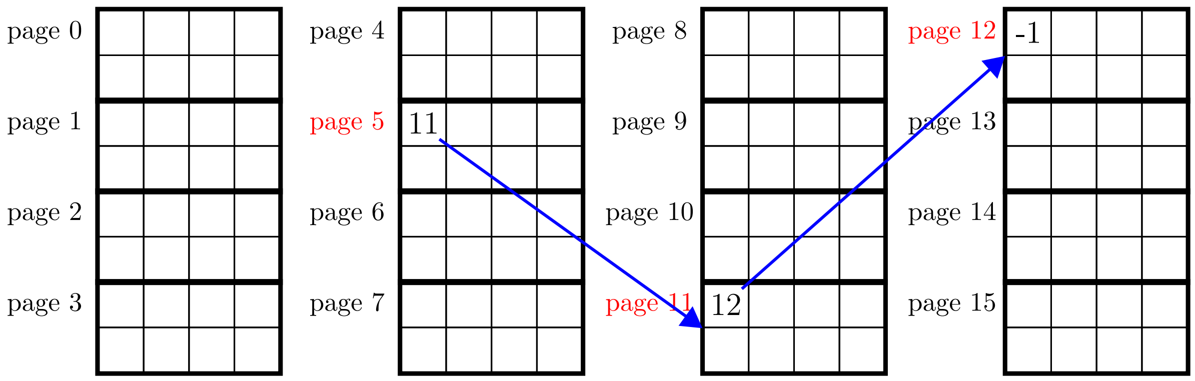
Note that we still need to address the issue of finding the first directory page for a relation. (And we also need to store the schema and metadata for that relation.) Before that we have to deal with free pages.

1.3.3 Free pages

Sometimes, you have a complete page that is not used. This happens for instance during initialization, during defragmentation of a database, and when a whole table/relation is deleted.

How do we manage these free pages?

VERSION 1. One option is to maintain a linked list of free pages. The next page number is stored in such pages. For instance if the first free page is page 5, then you might have something like this:



i.e., pages 5, 11, 12 are the free pages and they form a linked list Of course you can also make the free pages into a doubly linked list.

Note that the page number of the first page (the head of the singly linked list) has to be stored somewhere.

Exercise 1.3.1. Do you think the free pages should be linked up as a doubly linked list?

[Answer: The operations on the free pages are remove a free page to be used and to add a page to the free page. Therefore a singly linked list.]

1.3.4 Version 2.

You can also have a directory of pages for free pages. For instance, using the diagram above, suppose page 8 is the first of a directory page containing pointers to free pages. This is similar to the case of directory pages for records pages. The only difference is that for the directory page for records pages, you keep entries of (page number, size of free space)s while for the directory of free pagee, you have entries of page numbers.

Exercise 1.3.2. Compare Version 1 and Version 2 for the design of free pages.

[Answer: Version 1 can be arbitrarily long, limited only by the number of bytes allocated to the storage of page number. Version 2

Note. A heap file system is a storage system for records. It's entirely up to the RDBMS designer to use one heap file system for the whole database or to use one heap file for each table in the database.

1.3.5 Insert

Suppose you want to insert a record into a heapfile, say the heapfile for table Persons. There are several ways of doing that:

- Append: If you want to keep the records in the order of inserts, you can insert the record to the last page. If the last page does not have enough space, you request for a free page.
- Best fit: Iterate through all records using the doubly linked list of directory pages and look for a page with the best fit, i.e., the page with the minimal free space that can accommodate the new record This reduced fragmentation but can involve scanning the whole directory.

- First fit: Use the first page that contains enough free space. Fast, but early pages fill up quickly and slows down later inserts.
- Next fit: Maintain a running pointer that points to the last page where there was an insert. Iterate using this point while search for page for insert.

1.3.6 Delete

Assume you have the location of a record (say you have the rid), you just locate the slot in the directory and change the offset to -1.

Notes.

- If this is the last record in the page, you can also reclaim this space and put that into the free space (slot directory is updated by decrementing the array size). You would need to update the free space data in the directory page. This implies you somehow know the directory page number. Furthermore you must be able to find the location of this page in the directory. You can add directory page number into each records page. And the page numbers in the directory can be sorted by page numbers. Or: include both the directory page number and index position of the page in the directory page. This implies that the entries in the directory cannot be moved.

1.3.7 Metadata and schema

The following information is needed to work on our pages of records, pages of directories of records pages, pages of directories of free pages:

1. For each relation, we need to know what is the first directory page (the head of the doubly linked list)
2. For each relation, we need to know what is the structure of a record in this relation.
   a) Number of fixed width fields and their widths
   b) Number of variable length fields

   After reading the bytes for a field, we need to convert them to values of the right type. For instance for an `INT` field, we need to convert 4 bytes into an integer value.
3. We need to know what is the first directory page of the free pages.

It's important for now not to worry about database relations. Think of storing records of fields for now where fields are bytes.

Actually we can formulate the metadata as records. For instance suppose that you have a table T with 5 fields:

1. Field 1: fixed length, 4 bytes
2. Field 2: fixed length, 8 bytes
3. Field 3: fixed length, 2 bytes
4. Field 4: variable length, maximum 100 bytes
5. Field 5: variable length, maximum 200 bytes

This information can be translated into the following:

$$(3, 2, 4, 8, 2, 100, 200)$$

i.e.,

1. There are 3 fixed length fields and 2 variable length fields.
2. The fixed length fields have lengths 4, 8, 2 (in bytes).
3. The variable length fields have maximum lengths 100, 200 (in bytes).

The 3 and 2 above are fixed in length. The remaining (4, 8, 2, 100, 200) is variable in length. Therefore the metadata for T can be described as a record with 3 fields.

`Meta_Tables` relation:

1. `id`: `INT`
2. `name`: `VARCHAR(50)`
3. `firstdirpage`: `INT`

`Meta_Fields` relation:

1. `id`: `INT`
2. `name`: `VARCHAR(50)`
3. `table_id`: `INT`
4. `type`: `INT`
5. `length`: `INT`

where the `type` numbers are, for instance,

- `1` – `INT`
- `2` – `VARCHAR`
- `3` – `CHAR`
- `4` – `DATE`

etc. This is not standardized and is entirely up to the RDBMS vendor. In the case of variable length fields, `length` is the maximum length. The `table_id` points a `Fields` row to a row in `Tables`.