CS W186 Databases - Fall 2019
Guerilla Section 1: Disks, Files, B-Tree Indices

Sunday, September 15, 2019

Question 1: Files, Pages, Records

Consider the following relation:

```
CREATE TABLE Cats (
    collar_id INTEGER PRIMARY KEY, -- cannot be NULL!
    age INTEGER NOT NULL,
    name VARCHAR(20) NOT NULL,
    color VARCHAR(10) NOT NULL
);
```

You may assume that:

- `INTEGER`s are 4 bytes long;
- `VARCHAR(n)` can be up to n bytes long.

(a) As the records are variable length, we will need a *record header* in the record. How big is the record header? You may assume pointers are 4 bytes long, and that the record header only contains pointers.

Answer: **8 bytes.**
In the record header, we need *one pointer for each variable length record.* In this schema, those are just the two `VARCHAR`s, so we need 2 pointers, each 4 bytes.

(b) Including the record header, what is the smallest possible record size (in bytes) in this schema?

Answer: **16 bytes** (= 8 + 4 + 4 + 0 + 0)
8 for the record header, 4 for each of integers, and 0 for each of the `VARCHAR`s.

Note: There were a lot of questions about whether `VARCHAR() NOT NULL` meant that the field could not be the empty string.
`NULL` is treated as a special value by SQL, and an empty string `VARCHAR` is different from `NULL`, just like how a 0 `INTEGER` value is also different from `NULL`. You can test this for yourself with `psql` in your Docker container.
After reviewing the video lectures, we have determined that neither this, nor the topic of `NULL` value storage, were adequately communicated in lecture, and thus we will provide the necessary clarification should similar questions appear on an exam.

(c) Including the record header, what is the largest possible record size (in bytes) in this schema?

Answer: **46 bytes** (= 8 + 4 + 4 + 20 + 10)

(d) Now let's look at pages. Suppose we are storing these records using a slotted page layout with variable length records. The page footer contains an integer storing the record count and a pointer to free space, as well as a slot directory storing, for each record, a pointer and length. What is the **maximum** number of records that we can fit on a 8KB page? (Recall that one KB is 1024 bytes.)

Answer: **341 records** (= (8192 - 4 - 4) / (16 + 4 + 4))
We start out with 8192 bytes of space on the page.
We subtract 4 bytes that are used for the record count, and another 4 for the pointer to free space.
This leaves us with 8192 - 4 - 4 bytes that we can use to store records and their slots.
A record takes up 16 bytes of space at minimum (from the previous questions), and for each record we also need to store a slot with a pointer (4 bytes) and a length (4 bytes). Thus, we need 16 + 4 + 4 bytes of space for each record and its slot.

Note for guerilla attendees: The previous version of this question erroneously stated that slots only contained a pointer and not the length. For that version, an answer of **409** records would be considered correct.

(e) Suppose we stored the maximum number of records on a page, and then deleted one record. Now we want to insert another record. Are we guaranteed to be able to do this? Explain why or why not.

Answer: No, we deleted 16 bytes but the record we want to insert may be up to 46 bytes.

(f) Now suppose we deleted 3 records. Without reorganizing any of the records on the page, we would like to insert another record. Are we guaranteed to be able to do this? Explain why or why not.

Answer: No; there are 48 free bytes but they may be fragmented - there might not be 46 contiguous bytes.

Question 2: Files, Pages, Records

Consider the following relation:

```
CREATE TABLE Student (
    student_id INTEGER PRIMARY KEY,
    age INTEGER NOT NULL,
    units_passed INTEGER NOT NULL,
);
```

(a) Are `Student` records represented as fixed or variable length?

Answer: **Fixed.** There are no variable length fields (e.g. `VARCHAR`s).

Note: Some students asked about whether a `NULL`able field would make it variable length.
Records with `NULL`able fields can be represented either as fixed or variable length (see lecture DBF 1 → "Fixed and Variable Length Records").

In general, "fixed" vs. "variable" length is a property of the chosen representation of the record, and not necessarily the record itself. `VARCHAR(20)` could be represented by padding to exactly 20 bytes if we really wanted to (see the above lecture video for this as well).
However, records with only fixed-length fields admit only fixed-length representations, so the for this question, the answer is clear.

(b) To store these records, we will use an unpacked representation with a page header. This page header will contain nothing but a bitmap, rounded up to the nearest byte. How many records can we fit on a 4KB page?

Answer: **337** (= 4096 / (12 + 0.125))
The page must be divided up by 12 bytes per record plus 1 bit (0.125 bytes) for that record's bit in the bitmap.

(c) Suppose there are 7 pages worth of records. We would like to execute

```
SELECT * FROM Student WHERE student_id = 3034213355; -- just some number
```

Suppose these pages are stored in a heap file implemented as a linked list. What is the minimum and maximum number of page I/Os required to answer the query?

Answer: **Minimum: 2. Maximum: 8.**
First, 1 page read for the header.
The record can be on any of the 7 data pages; if we're lucky, we read 1 of them, and if we're unlucky, we read all 7.

(d) Now suppose these pages are stored in a sorted file, sorted on `student_id`. What is the minimum and maximum number of pages you would need to touch? You can assume sorted files do not have header pages.

Answer: **Minimum: 1. Maximum: 3.**
As seen in Lecture 5, we do binary search to find records in a sorted file.

Question 3: Files, Pages, Records

Note for guerilla section attendees: These questions have been edited for clarity.

(a) Suppose we are storing variable length records in a linked list heap file. In the "pages with space" list, suppose there happens to be 5 pages. What is the maximum number of page IOs required in order to insert a record?

You may assume that *at least one of these pages contains enough space*, and additionally that *it will not become full after insertion*.

Answer: **7 page I/Os.**
(1) Read header page.
(5) Read up to all 5 data pages (since the records are variable length, not all pages with space might have enough space to store a large record).
(1) Write updated data page (with the newly inserted record).

(b) Continuing from part (a), suppose on the contrary now that the page **does** become full after insertion. Now, we need to move that page to the "full pages" list.

Assume we have already done all necessary page reads for part (a)'s worst case (and that those pages are still in memory), but have not yet done any page writes.

How many **additional** page I/Os do we need to move the page to the "full pages" list?

Answer: **5 additional page I/Os.**
The idea behind the answer is this: We have two doubly linked lists; one for the full pages, another for the non-full pages; they are both connected to the header page.
We have a page at the end of the "non-full pages" list (from part (a)) that we would like to move to the head of the "full pages" list (the head, because that's the cheapest place to insert, which we are allowed to do because the pages are not linked in any particular order).
So - we are doing a doubly-linked list node movement from the tail of one list to the head of the other, which is done through pointer updates. We need to update pointers on the page we're moving, its old neighbour, and its two new neighbours (the header page, and the old first page in the full pages list).
Counting this out, this becomes:
(1) Write updated data page.
(1) Write previous non-full page (the old backwards neighbour).
(1) Read old first full page (the new forwards neighbour).
(1) Write old first full page.
(1) Write header page (the new backwards neighbour).

(c) Now suppose records are fixed length; what is the maximum number of page I/Os to insert a record?

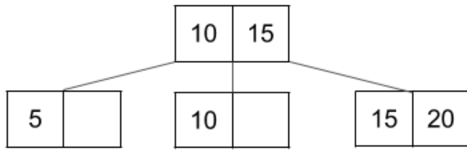
Answer: **3 page I/Os.**
(1) Read page header.
(1) Read a non-full page.
(1) Write updated non-full page. (This insert is guaranteed because this question says the records are fixed length.)

(d) Now suppose we are using a page directory, with one directory page. What is the maximum number of page I/Os we might have to do in order to insert a record?

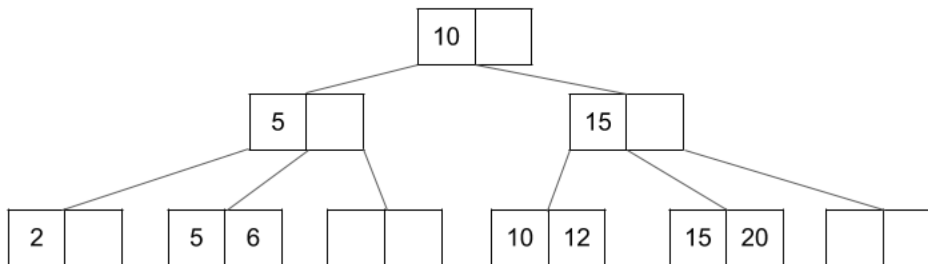
Answer: **4 page I/Os.**
(1) Read page header.
(1) Read a non-full page.
(1) Write updated non-full page.
(1) Write updated page header (we need to update the amount of free space).

Question 4: B-Trees

Given the following (degree $d = 1$) B+ tree:



(a) Draw what the tree looks like after adding 2, 6, and 12 in that order.



For the following questions, consider the tree directly after 2, 6, and 12 have been added.

(b) What is the maximum number of inserts we can do without changing the height of the tree?

Answer: **11 inserts**. The maximum number of keys for a fixed height h is given by $2d \cdot (2d+1)^h$. We have $d = 1$ from the question, and $h = 2$ (we must remember h is the number of non-root layers). Plugging these numbers in gives us $2 * 3^2 = 18$.
Now, we already have 7 keys in the tree, so we can insert $18 - 7 = 11$ more.

(c) What is the minimum number of inserts we can do that will change the height of the tree?

Answer: **4 inserts** (e.g. 16, 17, 18, 19).
The idea is to repeatedly insert into the fullest nodes; this is hopefully apparent from understanding how and when B-trees split.

Question 5: B-Trees

Consider the following schema:

```
CREATE TABLE FineWines (
    name CHAR(20) NOT NULL,
    years_aged INTEGER NOT NULL,
    price INTEGER NOT NULL
);
```

Suppose that we have built an index over `<years_aged, price>`, and suppose this index is two levels deep (in addition to the root node), and data is stored by *list of references* in separate data pages, each of which can hold hundreds of records.

(a)
```
SELECT * FROM FineWines
WHERE years_aged = 5
  AND price = 100
```

What is the worst case number of page I/Os to execute this query on an unclustered index if there is 1 matching record? 2 matching records? 3 matching records?

Answer: **4, 5, 6 page I/Os**.
As the B-tree is two levels deep, we must do 3 page reads to traverse the index tree: root node, layer 1 node, layer 2 (leaf) node. As the query is an *exact* index key match, and we are storing by *list of references*, all pointers to records for this key will be on the same leaf node.
However, since the index is unclustered, each record might be stored on its own page. Thus, we would need 1 additional page read for each matching record.

(b) What is the worst case number of page accesses to execute this query on a clustered index if there is 1 matching record? 2 matching records? 3 matching records?

Answer: **4, 5, 5 page I/Os**.
As before, the tree index traversal is 3 page I/Os. Since the index is now clustered, records are stored next to each other on pages.
However, there is no guarantee that they won't cross a page boundary. So, two matching records might be on two neighbouring pages: at the end of one, and at the beginning of another. Thus we need up to 2 additional page reads for ≥ 2 matching records.

(c)
```
SELECT * FROM FineWines
WHERE price = 100
```

What is the worst case number of page I/Os to execute this query if there are 150 data pages?

Answer: **150 page I/Os**.
Since this query *does not match* the available index, we just have to read every page.

(d)
```
DELETE FROM FineWines
WHERE years_aged = 1
  AND price = 5
```

Suppose this query deletes exactly one record. How many page I/Os are needed to execute this query? (Assume no tree rebalancing is required.)

Answer: **6 page I/Os**.
(3) Read index pages (traversing index tree). (1) Read data page. (1) Write data page. (1) Write leaf page in index (to delete the key from the index).