CS W186 - Spring 2020
Guerrilla Section 1
SQL, Disks and Files

Sunday, February 9, 2020

Question 1: SQL

Consider the following schema for library books:

```
Books: bid INTEGER,
       title TEXT,
       library REFERENCES Library,
       genre TEXT,
       PRIMARY KEY (bid)

Library: lid INTEGER,
         lname TEXT,
         PRIMARY KEY (lid)

Checkouts: book INTEGER REFERENCES Books,
           day DATETIME,
           PRIMARY KEY (book, date)
```

(a) Return the bid and genre of each book that has ever been checked out. Remove any duplicate rows with the same bid and genre.

```
SELECT DISTINCT b.bid, b.genre
FROM Books b, Checkouts c
WHERE b.bid = c.book
```

(b) Find all of the fantasy book titles that have been checked out and the date when they were checked out. Even if a book hasn't been checked out, we still want to output the title (i.e. the row should look like (title, NULL)).

```
SELECT title, day
FROM Books b LEFT OUTER JOIN Checkouts c ON c.book = b.bid
WHERE b.genre = 'Fantasy';
```

(c) Select the name of the book that has been checked out the most times and the corresponding checked out count. You can assume that each book was checked out a unique number of times, and that the titles of the books are all unique.

```
SELECT title, count(*) as cnt
FROM Books b, Checkouts c
WHERE b.bid = c.book
GROUP BY b.title
ORDER BY cnt DESC
LIMIT 1;
```

(d) Select the name of all of the pairs of libraries that have books with matching titles. Include the name of both libraries and the title of the book. There should be no duplicate rows, and no two rows that are the same except the libraries are in opposite order (e.g. ('East', 'West', 'Of Mice and Men') and ('West', 'East', 'Of Mice and Men')). To ensure this, the first library name should be alphabetically less than the second library name. **There may be zero, one, or more than one correct answer.**

(A)
```
SELECT DISTINCT l.lname, l.lname, b.title
FROM Library l, Books b
WHERE l.lname = b.library
AND b.library = l.lid
ORDER BY l.lname
```

(B)
```
SELECT DISTINCT l1.lname, l2.lname, b1.title
FROM Library l1, Library l2, Books b1, Books b2
WHERE l1.lname < l2.lname AND b1.title = b2.title
AND b1.library = l1.lid AND b2.library = l2.lid
```

(C)
```
SELECT DISTINCT first.l1, second.l2, b1
 FROM
    (SELECT lname l1, title b1
       FROM Library l, Books b
      WHERE b.library = l.lid) as first,
    (SELECT lname l2, title b2
       FROM Library l, Books b
      WHERE b.library = l.lid) as second
 WHERE first.l1 < second.l2
   AND first.b1 = second.b2;
```

Solution: B, C
A is incorrect because it does not return the books with matching titles but rather the names of the books at a specific library.
B is correct. C will perform a cross product of libraries with itself and books with itself matching books on title and matching that book to two different libraries.
C is correct. It performs two separate joins on libraries and books finding all of the book library pairs as inner subqueries. Then the outer query finds the matching book titles from the subqueries such that the library name for one is less than the second.

Question 2: More SQL

Consider the following schema for bike riders between cities:

```
Locations: lid INTEGER PRIMARY KEY,
           city_name VARCHAR

Riders: rid INTEGER PRIMARY KEY,
        name VARCHAR,
        home INTEGER REFERENCES locations (lid)

Bikes: bid INTEGER PRIMARY KEY
       owner INTEGER REFERENCES riders (rid)

Rides: rider INTEGER REFERENCES riders(rid)
       bike INTEGER REFERENCES bikes(bid)
       src INTEGER REFERENCES locations(lid)
       dest INTEGER REFERENCES locations(lid)
```

(a) Select all of the following queries which return the **rid** of **Rider** with the most bikes. Assume all **Riders** have a unique number of bikes.

(A) `SELECT owner FROM bikes GROUP BY owner ORDER BY COUNT(*) DESC LIMIT 1;`

(B)
```
SELECT owner FROM bikes GROUP BY owner HAVING COUNT(*) >= ALL
(SELECT COUNT(*) FROM bikes GROUP BY owner);
```

(C) `SELECT owner FROM bikes GROUP BY owner HAVING COUNT(*) = MAX(bikes);`

Solution: A, B
C is incorrect syntax MAX(bikes) does not make sense in this context.

(b) Select the **bid** of all **Bikes** that have never been ridden.

(A)
```
SELECT bid FROM bikes b1 WHERE NOT EXISTS
(SELECT owner FROM bikes b2 WHERE b2.bid = b1.bid);
```

(B)
```
SELECT bid FROM bikes WHERE NOT EXISTS
(SELECT bike FROM rides WHERE bike = bid);
```

(C)
```
SELECT bid FROM bikes WHERE bid NOT IN
(SELECT bike FROM rides, bikes as b2 WHERE bike = b2.bid);
```

Solution: B, C
A finds all of the bikes with no owners which is not the question we are trying to answer.

(c) Select the **name** of the rider and the **city_name** of the **src** and **dest** locations of all their journeys for all rides. Even if a rider has not ridden a bike, we still want to output their name (i.e. the output should be (name, null, null)).

(A)
```
SELECT tmp.name, s.city_name AS src, d.city_name AS dst FROM
locations s, locations d,
(riders r LEFT OUTER JOIN rides ON r.rid = rides.rider) as tmp
WHERE s.lid = tmp.src AND d.lid = tmp.dest;
```

(B)
```
SELECT r.name, s.city_name AS src, d.city_name as dst FROM riders r
LEFT OUTER JOIN rides ON r.rid = rides.rider
INNER JOIN locations s on s.lid = rides.src
INNER JOIN locations d on d.lid = rides.dest;
```

(C)
```
SELECT r.name, s.city_name AS src, d.city_name AS dst FROM rides
RIGHT OUTER JOIN riders r ON r.rid = rides.rider
INNER JOIN locations s on s.lid = rides.src
INNER JOIN locations d on d.lid = rides.dest;
```

Solution: None of these are correct, because the subsequent inner joins/where clauses will filter out the null rows from the outer join.

Question 3: 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 value.* 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? (Note: `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`. We will provide the necessary clarification should similar questions appear on an exam.)

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.

(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.

(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 4: 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 5: Files, Pages, Records

(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? Assume that the page we insert into does not fill up after the insertion.

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).