CS 186 Exam Prep Section 1:
SQL, Disks and Files

Agenda (All times Pacific)

I. 5:10-5:30 PM -- Mini-lecture
II. 5:30-6:30 PM-- Worksheet
III. 6:30-7 PM-- Go over answers

Worksheet and slides can be found @89 on Piazza
Solutions will be released afterward

Feedback Form: **https://tinyurl.com/CS186ExamPrep1FA20**

SQL

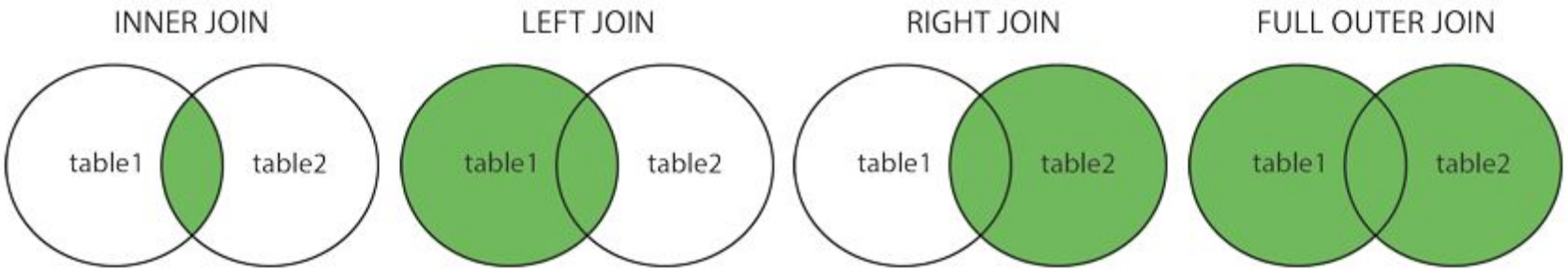
SQL Basics

```
SELECT [DISTINCT] <column list>
FROM <table1>
[WHERE <predicate>]
[GROUP BY <column list>]
[HAVING <predicate>]
[ORDER BY <column list> [DESC/ASC]]
[LIMIT <amount>];
```

Logical Processing Order

1. `FROM <table1>` - which table are we drawing data **from**
2. `[WHERE <predicate>]` - keep only rows **where** `<predicate>` is satisfied
3. `[GROUP BY <column list>]` - **group** together rows **by** value of columns in `<column list>`
4. `[HAVING <predicate>]` - keep only groups **having** `<predicate>` satisfied
5. `SELECT <column list>` - **select** columns in `<column list>` to keep
    a. `[DISTINCT]` - keep only **distinct** rows (filter out duplicates)
6. `[ORDER BY <column list> [DESC/ASC]]` - **order** the output **by** value of the columns in `<column list>`
7. `[LIMIT <amount>]` - **limit** the output to just the first `<amount>` rows

Joins



- How to use (people and boats are tables):
  - ...FROM people [INNER JOIN | LEFT JOIN | RIGHT JOIN | FULL JOIN] boats on people.id = boats.id …
- Default is inner join (...FROM people, boats WHERE people.id = boats.id…)
- NULL used to fill in entries with no matching data for non-inner joins
- If there's no join condition, does cartesian product (all possible pairs of rows, where each pair has one row from the first set and one row from the second set

String Comparison

- Note: not tested very heavily, but will probably show up
- LIKE: following expression follows SQL specified format
  - Looks for matches at start of the string
  - %: zero, one, or multiple characters
  - _: one character
- ~ : following expression follows regex format
  - Looks for matches anywhere in the string
  - . : Wildcard character
  - * : Allows for zero, one, or more of the character preceding the symbol
  - ^: Only match at the beginning of the string (functions like LIKE)

More SQL Things

- Sets - a collection with no duplicates
  - UNION - all items in either set, INTERSECTION - all items in both sets.
  - In SQL, these don't have to be used on sets (there can be duplicates in the inputs). No duplicates in the final set!
  - Must add ALL to include duplicates - UNION ALL, INTERSECTION ALL
  - Used between two queries (SELECT … FROM … UNION SELECT … FROM ...)
- Correlated Query
  - Not emphasized, but tends to come up on exams
  - Subquery depends on values from the outer query, subquery is recalculated for every row of the outer query's table
  - SELECT S.sname FROM Sailors S WHERE EXISTS (SELECT * FROM Reserves R WHERE R.bid=102 AND S.sid=R.sid)

More SQL Things

- Aggregations - COUNT, SUM, AVG, etc.
  - Using an aggregate in **WHERE** is **not** allowed! WHERE **count(*) > 500** is an **invalid** query!
  - NULL column values are ignored by aggregate functions
- Don't use **HAVING** without **GROUP BY - Just use **WHERE** instead!
- **DISTINCT** removes all **duplicate rows**

Three-valued logic:

NOT	T	F	N
	F	T	N

AND	T	F	N
T	T	F	N
F	F	F	F
N	N	F	N

OR	T	F	N
T	T	T	T
F	T	F	N
N	T	N	N

Disks & Files

Files, Pages, 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)
- The unit of access to physical disk is the page
  - 1 I/O = read or write 1 page

Page Basics:

The **page header** keeps track of the records in the page.

The page header may contain fields such as:

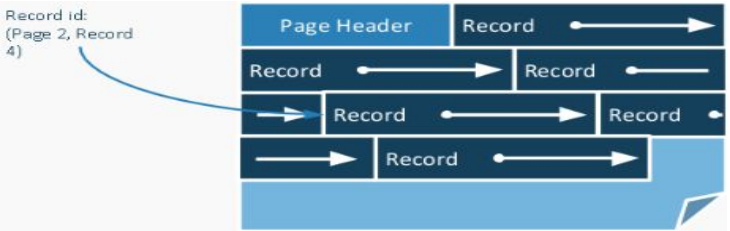
- Number of records in the page
- Pointer to segment of free space in the page
- Bitmap indicating which parts of the page are in use

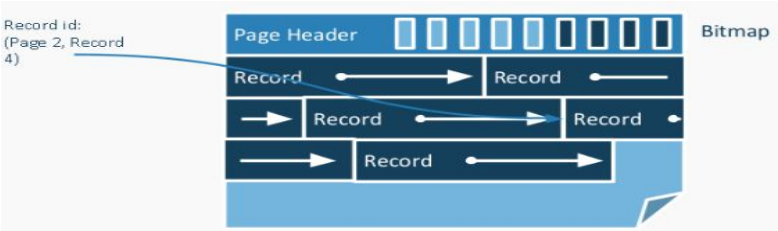
Fixed Length Records

Fixed length records are when **record lengths are fixed** and **field lengths are consistent**

Packed Records: no gaps between records, record ID is location in page

Unpacked Records: allow gaps between records, use a bitmap to keep track of where the gaps are

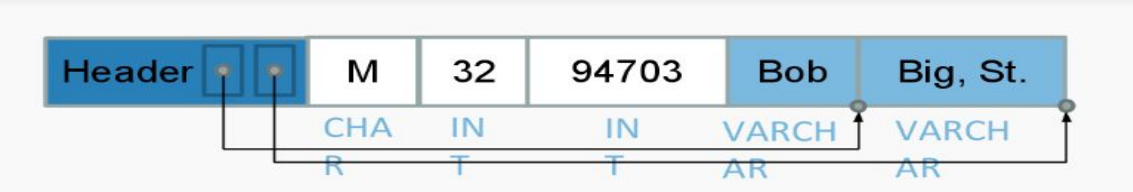




Variable Length Records

Variable length records are when either **record lengths are not fixed** or **field lengths are not consistent**

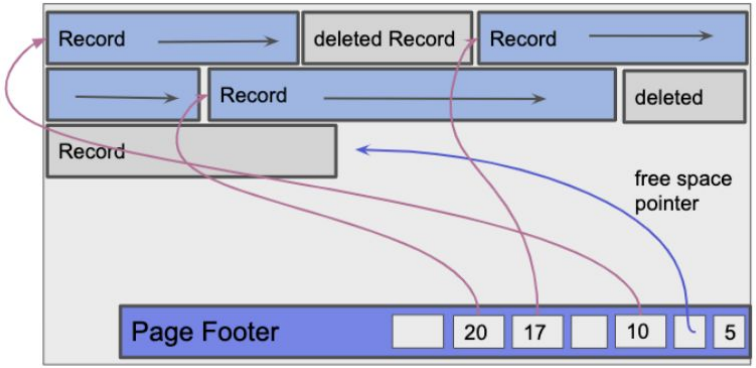
We can store variable length length records with an array of field offsets:



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

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 pointer to free space
- Deleting records may cause fragmentation if we use an *unpacked* layout



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

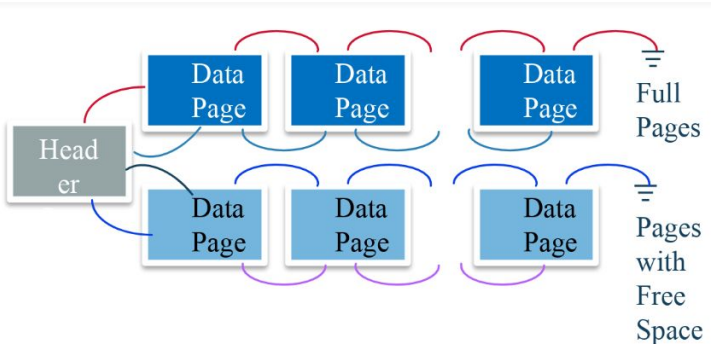
A **sorted file** is similar to a heap file, except we require it be sorted on a key (a subset of the fields).

- Implemented using page directories, data pages ordered based on how records are sorted
  - We can use binary search on the key the file is sorted on to find records!

Heap Files: List Implementation

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

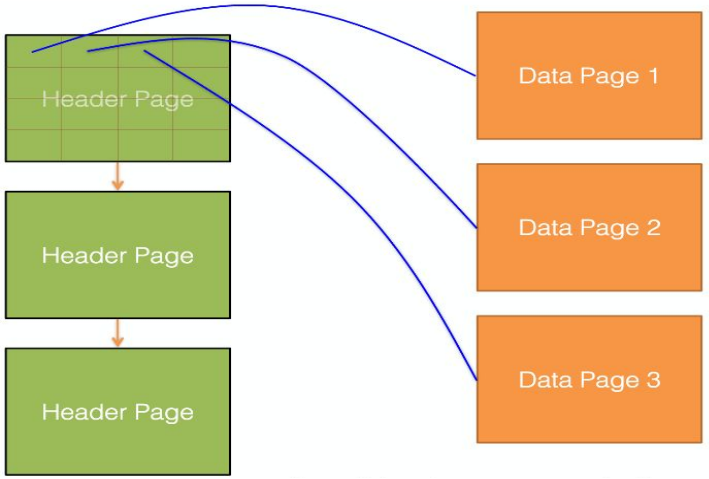


Heap Files: Page Directory Implementation

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



Operations: Heap Files vs Sorted Files

- Scan: read all of the pages of the file
  - Same for heap files and sorted files
- Equality Search: find all records in the file matching a certain key
  - Heap files: complicated equation, but expect to touch about half the pages in the file
  - Sorted files: do binary search to find the key
- Range Search: find all records in the file whose key is in a given range
  - Heap files: scan whole file
  - Sorted files: do binary search to find the lower bound key, then scan for all records with keys in the range
- Insert: add a record to the file
  - Heap files: add it to the end of a page with free space (or write a new page if all pages are full)
  - Sorted files: do binary search to find where record should go, then shift all of the records after it
- Delete: remove a record from the file
  - Heap files: touch about half of the pages in the file to find the record (see equality search), then write to the page to remove
  - Sorted files: do binary search to find the record, remove, then shift all of the records after it

Operations: Heap Files vs Sorted Files

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D
Range Search	B*D	$((\log_2 B)$+pages)* D
Insert	2*D	$((\log_2 B)$+B)*D
Delete	(0.5*B+1)*D	$((\log_2 B)$+B)*D

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Worksheet

@89 on Piazza

Solutions

Question 1

Question 1a

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 bid, genre
FROM Books B, Checkouts C
WHERE B.bid = C.book;
```

Need to join **Books** and **Checkouts** to get genre and the fact that a book is checked out

INNER JOIN would work as well

```
CREATE TABLE Books (
    bid INTEGER,
    title TEXT,
    library REFERENCES Library,
    genre TEXT,
    PRIMARY KEY (bid)
);

CREATE TABLE Library (
    lid INTEGER,
    lname TEXT,
    PRIMARY KEY (lid)
);

CREATE TABLE Checkouts (
    book INTEGER REFERENCES Books,
    day DATETIME,
    PRIMARY KEY (book, date)
);
```

Question 1b

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)).

Wanting **NULL** in the output is a good sign of a LEFT OUTER or RIGHT OUTER join

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

LEFT JOIN to preserve all books; use WHERE to filter genre of the joined table

```
CREATE TABLE Books (
    bid INTEGER,
    title TEXT,
    library REFERENCES Library,
    genre TEXT,
    PRIMARY KEY (bid)
);

CREATE TABLE Library (
    lid INTEGER,
    lname TEXT,
    PRIMARY KEY (lid)
);

CREATE TABLE Checkouts (
    book INTEGER REFERENCES Books,
    day DATETIME,
    PRIMARY KEY (book, date)
);
```

Question 1c

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

Want title from **Books** table, # of checked out from aggregating on **Checkouts.**
Group by title, not bid --- bid is unique for each “instance” of a book!

```
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;
```

```
CREATE TABLE Books (
    bid INTEGER,
    title TEXT,
    library REFERENCES Library,
    genre TEXT,
    PRIMARY KEY (bid)
);

CREATE TABLE Library (
    lid INTEGER,
    lname TEXT,
    PRIMARY KEY (lid)
);

CREATE TABLE Checkouts (
    book INTEGER REFERENCES Books,
    day DATETIME,
    PRIMARY KEY (book, date)
);
```

Question 1d

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. To ensure this, the first library name should be alphabetically less than the second library name. T**here 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;
```

Incorrect: does not return books with matching titles. Also incorrectly uses lname to compare to **Books.**library

```
-- 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;
```

Correct: Filters rows in cross join where the alphabetic order is respected and book titles are the same

```
-- 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;
```

Correct: Finds book-library pairs as inner subqueries. Outer query does the cross join of these pairs such that the titles are the same and l1 is alphabetically “less than” l2

Question 2

Question 2a

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

```
CREATE TABLE Locations (
    lid INTEGER PRIMARY KEY,
    city_name VARCHAR
);

CREATE TABLE Riders (
    rid INTEGER PRIMARY KEY,
    name VARCHAR,
    home INTEGER REFERENCES location (lid)
);

CREATE TABLE Bikes (
    bid INTEGER PRIMARY KEY,
    owner INTEGER REFERENCES riders (rid)
);

CREATE TABLE Rides (
    rider INTEGER REFERENCES riders (rid),
    bike INTEGER REFERENCES bikes (bid),
    src INTEGER REFERENCES location (lid),
    dst INTEGER REFERENCES location (lid)
);
```

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

Correct: owner references rid so we don't need to join on another table. Just aggregate!

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

Correct: returns "all" owners that have at least as many bikes as all owners. Because all riders have a unique # of bikes, this returns the 1 rider with the most bikes

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

Incorrect: using MAX on the table **bikes** is nonsensical (what even would be aggregated?)

Question 2b

Select the **bid** of all **Bikes** that have never been ridden

```
CREATE TABLE Locations (
    lid INTEGER PRIMARY KEY,
    city_name VARCHAR
);

CREATE TABLE Riders (
    rid INTEGER PRIMARY KEY,
    name VARCHAR,
    home INTEGER REFERENCES location (lid)
);

CREATE TABLE Bikes (
    bid INTEGER PRIMARY KEY,
    owner INTEGER REFERENCES riders (rid)
);

CREATE TABLE Rides (
    rider INTEGER REFERENCES riders (rid),
    bike INTEGER REFERENCES bikes (bid),
    src INTEGER REFERENCES location (lid),
    dst INTEGER REFERENCES location (lid)
);
```

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

Incorrect: Finds all bikes without an owner --- not what we want!

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

Correct: Finds all bikes in the **Bikes** table for which there are no entries in the **Rides** table

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

Correct: Finds all bikes in the **Bikes** table that do not exist in the join of **Rides** and **Bikes**

Question 2c

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.

```
-- 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;
```

None of these are correct!

The INNER JOINs and WHERE clauses will filter out rows with NULL values produced by the OUTER JOIN.

Question 2c

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.

How would we construct a correct answer?

Need to do a JOIN on **Locations** to get the city_name, but need to do this *before* the join onto riders

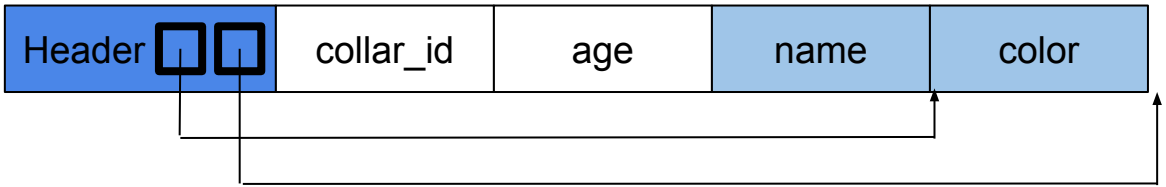
```
SELECT riders.name, city_rides.src, city_rides.dst
FROM riders LEFT OUTER JOIN (
    SELECT rides.rider as rid, s.city_name as src, d.city_name as dst
    FROM rides, locations s, locations d
    WHERE rides.src = s.lid
      AND rides.dest = d.lid
) city_rides ON riders.rid = city_rides.rid;
```

Question 3

Question 3a

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.

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



Question 3a

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.

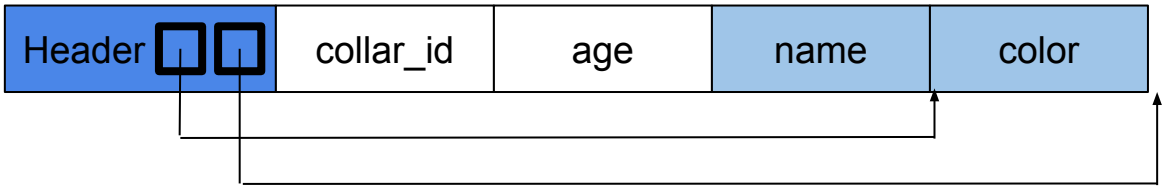
```
CREATE TABLE Cats (
    collar_id INTEGER PRIMARY KEY,
    age INTEGER NOT NULL,
    name VARCHAR(20) NOT NULL,
    color VARCHAR(10) NOT NULL
);
```

8 bytes. In the record header, we need one pointer for each variable length value. In this schema, those are just the two VARCHARs, so we need 2 pointers, each 4 bytes.

Question 3b

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

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



Question 3b

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

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

Question 3b

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

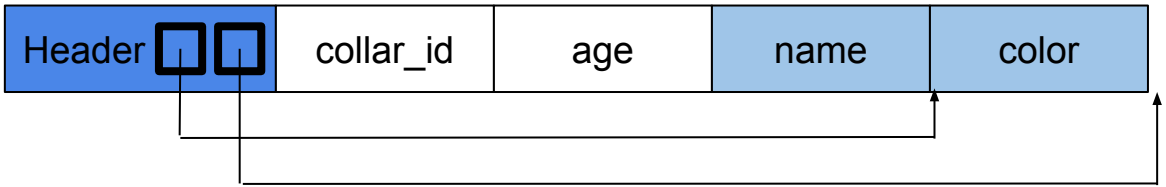
```
CREATE TABLE Cats (
    collar_id INTEGER PRIMARY KEY,
    age INTEGER NOT NULL,
    name VARCHAR(20) NOT NULL,
    color VARCHAR(10) NOT NULL
);
```

16 bytes (= 8 + 4 + 4 + 0 + 0). 8 for the record header, 4 for each of integers, and 0 for each of the VARCHARs.

Question 3c

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

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



Question 3c

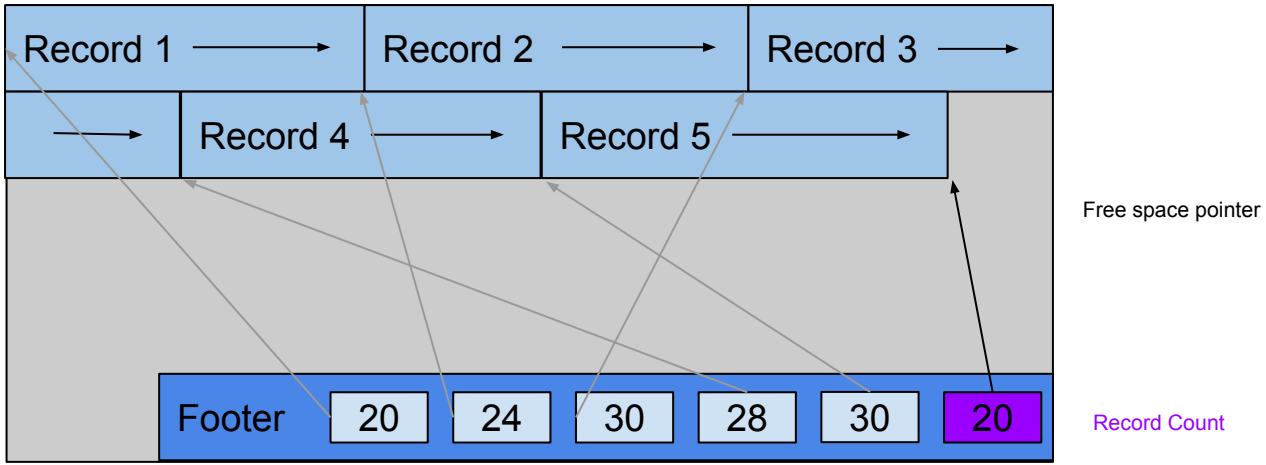
Including the record header, what is the largest possible record size (in bytes) in this schema?

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

46 bytes (= 8 + 4 + 4 + 20 + 10)

Question 3d

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?



Question 3d

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?

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.

Question 3e

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.

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

Question 3f

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.

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

Fill out our feedback form!

https://tinyurl.com/CS186ExamPrep1FA20