CS W186 Spring 2019 Midterm 1

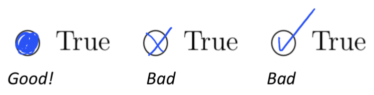
Do not turn this page until instructed to start the exam.

Contents:

- You should receive one *double-sided answer sheet* and a 17-page *exam packet*.
- The midterm has *5 questions*, each with multiple parts.
- The midterm is worth a total of *60 points*.

Taking the exam:

- You have *110 minutes* to complete the midterm.
- All answers should be written on the answer sheet. The exam packet will be collected but not graded.
- For each question, place only your *final answer* on the answer sheet; do not show work.
- For multiple choice questions, please *fill in the bubble or box completely* as shown on the left below. ***Do not mark the box with an X or checkmark***.



- Use the blank spaces in your exam for scratch paper.

Aids:

- You are allowed **one** 8.5" × 11" double-sided pages of notes.
- The **only** electronic devices allowed are basic scientific calculators with simple numeric readout. No graphing calculators, tablets, cellphones, smartwatches, laptops, etc.

Grading Notes:

- All IOs must be written as integers. There is no such thing as 1.04 IOs – that is actually 2 IOs.
- 1 KB = 1024 bytes. We will be using powers of 2, not powers of 10
- Unsimplified answers, like those left in log format where simplification to integers is possible, will receive a point penalty.

1 Sorting and Hashing (14 points)

For this question, we consider the following relation:

```
CREATE TABLE students (
    sid INTEGER PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    graduation_year INTEGER NOT NULL,
    favorite_number INTEGER NOT NULL
);
```

- `sid` is an integer ranging from `20000000-29999999`.
- `graduation_year` is one of `2019, 2020, 2021, 2022`.
- `favorite_number` can be any 32-bit integer.

Unless otherwise stated, assume that all hash functions used partition data evenly.

1. (2 points) If we have 100 pages of memory, and the `students` relation is 186,000 pages large, how many I/Os are required to sort the relation on `sid`? Include the final write in your answer.

   Solution: 1,116,000 I/Os. 2 merge passes are required (we have 1,860 runs before merging).

2. (2 points) If we have 10,000 pages of memory, and the `students` relation is 186 pages large, how many I/Os are required to sort the relation on `favorite_number`? Include the final write in your answer.

   Solution: 372 I/Os. We can do this in memory and no merge passes are required.

3. (2 points) If we have 10 pages of memory, and the `students` relation is 200 pages large, how many I/Os are required to hash the relation on `sid`? Include the final write in your answer.

   Solution: 1343 I/Os.

   We partition 200 pages into B - 1 = 10 - 1 = 9 partitions. $\lceil 200/9 \rceil = 23$, so each partition is 23 pages. We need to recursively partition all of these partitions again, since they are all too large to fit in our buffer (they need to be at most B = 10 pages large).

   After recursively partitioning a 23-page partition, we get $\lceil 23/9 \rceil = 3$ pages per partition. This will definitely fit into our buffer, so we can read in these partitions, build in memory hash tables out of them, and then write the hash tables to disk.

   The first partitioning step reads in 200 pages and writes $23 \cdot 9 = 207$ pages.

   The second (recursive) partitioning step reads in 207 pages and writes $3 \cdot 9 \cdot 9 = 243$ pages.

   The hash table building step reads in 243 pages and writes 243 pages.

   200 + 207 + 207 + 243 + 243 + 243 = 1343 I/Os.

   Note: originally the answer was 1200 from using the formula, but starting in Fall 2019, using the updated hashing method, the answer should be as above.

4. (1 point) If we have B pages of memory, and the `students` relation is $B + 1$ pages large, how many hash functions do we need to hash `students` on `sid` using the external hashing algorithm?

Solution: 2. We need one for partitioning and one for probing (assuming a perfect hash function, we would have partitions of size 2 after the first pass).

5. (1 point) If we have B pages of memory, and the `students` relation is B^2 pages large, how many hash functions do we need to hash `students` on `sid`?

Solution: 3. We need two for partitioning and one for probing (we get partitions of size $\frac{B^2}{B-1} > B$ after one pass, and partitions of size $\frac{B^2}{(B-1)^2}$ after two passes, which is at most B for $B \geq 3$ - the minimum required amount of memory for hashing).

6. (3 points) If we have 10 pages of memory, what is the **minimum** size in pages that `students` must be to need at least two partitioning passes when hashing on `sid, name` (i.e. at least three passes over the data)?

Solution: 91 pages. We can hash up to $(10 - 1) \cdot 10 = 90$ pages with only one partitioning pass.

7. (3 points) If we have 10 pages of memory, what is the **maximum** size in pages that `students` can be while only requiring at most two partitioning passes when hashing on **`graduation_year`**? (Note the hash key!)

Solution: 40 pages. We are hashing on `graduation_year`, which only has 4 distinct values. Therefore, under the assumption that the hash function distributes everything evenly, we effectively have 4 partitions after one partitioning pass. Recursive partitioning actually does nothing here, and we can only process relations of at most $4 \cdot 10 = 40$ pages.

2 Disks, Files, Pages & Records (8 points)

Consider the following table schema:

```
CREATE TABLE Dogs (
    dog_id INTEGER PRIMARY KEY,
    age INTEGER NOT NULL,
    name VARCHAR(15) NOT NULL
);
```

1. (2 points) Given that we are using a slotted page layout with variable length records, what is the **maximum** number of records we can fit on a 4KB page?

   - Assume integers and pointers are 4 bytes each.
   - Assume each page's footer contains an integer for the record count and a pointer to free space, as well as a slot directory that uses integers to store the length and pointer for each record.
   - Assume records have record headers containing pointers to the ends of the variable-length fields in the records.

Solution: 204 records

The page footer will have 8 bytes (for the record counter + free space pointer) as well as the slot directory

Each record will take up at minimum (8 + 4 + 8) = 20 bytes:

- 8 bytes: For the slot in the directory (4 bytes for the length/offset of the record + 4 bytes for the pointer to the beginning of the record)
- 4 bytes: For the pointer to the variable length field in the record header
- 8 bytes: For the 2 ints for dog id and age (0 bytes for name because were trying to find the maximum number of records that can fit on the page)

Thus, the total calculation for the 4KB page comes out to be (4 * 1024 - 8) / (20) = 204.4 $\rightarrow$ 204 records

For the following questions, consider the following table schema:

```
CREATE TABLE Teams (
    team_id INTEGER PRIMARY KEY,
    name CHAR(20) NOT NULL,
    region CHAR(20) NOT NULL,
    num_players INTEGER NOT NULL
);
```

2. (4 points) Given that the file contains 16 data pages and counting each read or write of a page as 1 I/O, calculate the **worst case** number of I/Os it would take to complete the following 3 queries.

   - Assume the file layout consists of a heap file with a page directory of 2 pages, where each page in the page directory can hold pointers to ten different data pages.
   - Assume the buffer is big enough to hold the page directory and all data pages.

- Assume operations are independent; i.e query 2 is run on a copy of the file that has never had query 1 run on it.
- Assume that to access each data page, you need to access its corresponding entry in the page directory. (e.g. you can't scan all 16 data pages by following sibling pointers between data pages)

(a) `SELECT * From Teams WHERE num_players > 20 AND num_players < 40;`

(b) `INSERT INTO Teams Values (404, "Not Found", "Unknown", 0);`

(c) `SELECT * From Teams WHERE team_id = 3;`

Solution:

1. 2 + 16 = 18 I/Os
   - 2: Access header pages
   - 16: Full scan of file
2. 2 + 1 + 1 + 1 = 5 I/Os
   - 2: Access header pages. Find there's enough free space to insert a record after accessing the second header page
   - 1: Read the page
   - 1: Insert record on page and write this back to disk
   - 1: Update the page directory and write this back to disk

   Alternate Solution:
   A student pointed out that you would need to make sure that the primary key is not already in the table. The total IOs are now:
   16 + 2 + 1 + 1 = 20 I/Os
   - 16: Data page reads to make sure the primary key is not already in the table
   - 2: Header page reads (part of the full scan)
   - 1: Write the updated data page (or the new data page if they were all full)
   - 1: Write the header page that has the record for the updated data page
3. 18 I/Os; just like (a), a full scan is needed for this, and you need to read in the header pages as well.

3. (2 points) Now, disregard the page directory. For each SQL statement above, select the scheme for storing the `Teams` table that would maximize speed in the average case: A heap file, with each page $\frac{2}{3}$ full, or a packed sorted file, where the sorted file is sorted on the primary key. If the two would take the same time, select the "Same" option on the answer sheet.

Solution:

1. Sorted: The sorted file is not sorted on num players, so the query reduces to a full scan for both heap files and sorted files, but as the heap file is 2/3 full while the sorted file is packed, the full scan of the heap file will cover more pages and will thus take longer.

2. Heap: Inserting into a heap file is simply at the end of the file. Inserting into a sorted file first requires finding where to insert the record into, and then shifting all the subsequent records.

3. Sorted; as the filter is on a primary key, which the file is sorted by, a sorted file is clearly faster for lookup.

3 Query Languages (12 points)

For this question, we consider the following relations:

```
CREATE TABLE Contestant (
    id      INTEGER PRIMARY KEY,
    name    TEXT,
    team    TEXT,
    points  INTEGER
);
CREATE TABLE TeamLeader (
  id            INTEGER REFERENCES Contestant(id),
  time TIMESTAMP
);
CREATE TABLE BonusPoint (
  id             INTEGER REFERENCES Contestant(id),
  time_collected TIMESTAMP
);
CREATE TABLE Alliance (
  team1      TEXT REFERENCES Contestant(team),
  team2      TEXT REFERENCES Contestant(team),
  start_time TIMESTAMP
);
```

- Every team is represented in `TeamLeader` – possibly more than once. The current leader is the one with the latest time stamp. (You should assume that whenever a team gets a new leader, we simply add a row to the table; we never remove any rows.)
- Assume that `Alliance` is symmetric. If (Blue, Gold, 12:00 AM) is a row in `Alliance` then so is (Gold, Blue, 12:00 AM).
- Alliances are not transitive: if Team 1 is allied with Team 2, and Team 2 is allied with Team 3, then Team 1 may or may not be allied with Team 3.

For concreteness, here are some examples of what the `Contestant` and `TeamLeader` tables may look like:

Contestant

id	name	team	points
3	Rex	Red	5
4	Woody	Blue	4
5	Buzz	Blue	1
6	Rex	Green	6
7	Zurg	Red	2

TeamLeader

id	time
3	2019-01-01 00:01:00
4	2019-01-01 00:02:00
6	2019-01-01 00:03:00
7	2019-01-01 00:04:00
5	2019-01-01 00:05:00

1. (2 points) Find the name of all Contestants that were ever promoted to TeamLeader, and their team name. If there are team leaders with duplicate names, these should show up multiple times. Mark True if the query is correct, False otherwise.

   A.
   ```
   SELECT C.name, C.team
     FROM Contestant C, TeamLeader T
    WHERE C.id = T.id;
   ```
   B.
   ```
   SELECT C.name, C.team
     FROM Contestant C NATURAL JOIN TeamLeader T;
   ```
   C.
   ```
   SELECT C.name, C.team
     FROM Contestant C RIGHT JOIN TeamLeader T ON C.id = T.id;
   ```
   D.
   ```
   SELECT C.name, C.team
     FROM Contestant C INNER JOIN TeamLeader T ON C.id = T.id;
   ```

2. (3 points) A bonus point is worthless, unless it was collected by a contestant whose team was in at least one alliance when the bonus point was collected. Compute how many valid bonus points each team has. Omit teams with no valid bonus points. Mark True if the query is correct, False otherwise.

   A.
   ```
   SELECT C.team, COUNT(*)
   FROM Contestant C, BonusPoint BP, Alliance A
   WHERE C.id = BP.id
     AND C.team = A.team1
     AND A.start_time <= BP.time_collected
   GROUP BY C.team;
   ```
   B.
   ```
   SELECT T1.team, COUNT(*)
   FROM (
     SELECT C.team, BP.time_collected AS time
     FROM Contestant C, BonusPoint BP
     WHERE C.id = BP.id
   ) AS T1, (
     SELECT A.team1 AS team, MIN(A.start_time) AS time
     FROM Contestant C, Alliance A
     GROUP BY A.team1
   ) AS T2
   WHERE T1.team = T2.team
     AND T2.time <= T1.time
   GROUP BY T1.team;
   ```
   C.
   ```
   SELECT C.team, COUNT(*)
   FROM Contestant C, BonusPoint BP
   WHERE C.id = BP.id
     AND EXISTS (
       SELECT *
       FROM Alliance A
       WHERE C.team = A.team1
       AND A.start_time <= BP.time_collected
     )
   GROUP BY team;
   ```

Solution: B, C

A. False - This query re-counts the bonus points collected by competitors who are in teams that appear multiple times in the `Alliance` table.

B. True - The first subquery finds, for each team, the time at which each bonus point was collected. The second subquery finds, for each each team, the time at which it first joined an alliance. We join these tables, and count the bonus points which were collected after the team in question first joined an alliance.

C. True - We match team to each row in the `BonusPoint` table and then check that there existed an alliance at the time of collection.

3. (2 points) Referring to the example tables provided before (and copied below), match each output below to the query that generated it.

Contestant

id	name	team	points
3	Rex	Red	5
4	Woody	Blue	4
5	Buzz	Blue	1
6	Rex	Green	6
7	Zurg	Red	2

TeamLeader

id	time
3	2019-01-01 00:01:00
4	2019-01-01 00:02:00
6	2019-01-01 00:03:00
7	2019-01-01 00:04:00
5	2019-01-01 00:05:00

Output 1

Blue	5
Red	7
Green	6

Output 2

Blue	6
Red	9
Green	12

Output 3

Blue	4
Red	5
Green	6

Output 4

ERROR

Query (a):

```
SELECT team, SUM(points)
FROM Contestant
WHERE points > 3
GROUP BY team;
```

Query (b):

```
SELECT team, MAX(points)
FROM Contestant;
```

Query (c): (Ignore this query. This question was thrown out.)

```
SELECT C.team, SUM(points) + lead_points
FROM Contestant C, (
  SELECT team, points AS lead_points
  FROM Contestant C, TeamLeader TL
  WHERE C.id = TL.id
    AND TL.time >= ALL(SELECT time FROM TeamLeader TL WHERE C.team = TL.team)
) T
WHERE C.team = T.team
GROUP BY C.team, lead_points;
```

Query (d):

```
SELECT C1.team, C1.id
FROM Contestant C1, TeamLeader TL1
WHERE C1.id = TL1.id
  AND TL1.time >= ALL(
    SELECT time
    FROM Contestant C2, TeamLeader TL2
    WHERE C2.id = TL2.id
      AND C1.team = C2.team
  );
```

Solution:

Query (a) - Output 3. This query computes each team's total score, accounting only for players who have more than 3 points.

Query (b) - Output 4. It's not legal to aggregate in this context.

Query (c) - Output 2. This query computes each team's total score, double-counting the current team leader's score.

Query (d) - Output 1. This query finds the id of the current team leader for each time.

4. (2 points) Consider the relations: `A(c1 PRIMARY KEY, c2, c3)` and `B(c1 PRIMARY KEY, c2, c3)`. Evaluate the claims below; mark True for each that is correct, False for each that is incorrect.

   A. `A FULL OUTER JOIN B ON A.c2 = B.c2` has the same number of rows as `A INNER JOIN B ON A.c2 = B.c2` if `A.c2` contains all the values that `B.c2` contains.

   B. There is no difference between the following queries:

```
SELECT c1            SELECT c1             SELECT c1
FROM A               FROM A                FROM A
ORDER BY c1 DESC     WHERE c1 >= ALL(      WHERE c1 = (
LIMIT 1;               SELECT c1             SELECT MAX(c1)
                       FROM A                FROM A
                     );                    );
```

Solution: B

A. False - `B.c2` must also contain all the values that `A.c2` contains.

B. True - `c1` is a primary key so duplicate rows are not a concern. Otherwise, these queries would be different.

For the following question, assume that relations can be referenced by their first letter. For example, we reference `Contestant` as C.

5. (3 points) Find the id and name of every contestant who at one point was or is currently a team leader and has scored at least one bonus point (including invalid bonus points). Mark True if the relational algebra expression is correct, False otherwise.

   A. $\pi_{\text{id, name}}(C \bowtie (\pi_{\text{name}}(C \bowtie B) \cap \pi_{\text{name}}(C \bowtie T)))$

   B. $\pi_{\textbf{id, name}}(C \bowtie B) - (\pi_{\textbf{id, name}}(C \bowtie B) - \pi_{\textbf{id, name}}(C \bowtie T))$

   C. $\pi_{\textbf{id, name}}(C \bowtie B) \cap \pi_{\textbf{id, name}}(C \bowtie T)$

Solution: A. False - Projecting only the names in the inner expression results in an incorrect query in the case of duplicate names.

B. True - $S_1 \cap S_2 = S_1 - (S_1 - S_2)$.

C. True - This is the correct intersection of the two sets we are looking for.

4 B+ Trees (13 points)

1. (5 points) Which of the following statements are true? **There may be zero, one, or more than one correct answer.**

   A. Indices allow us to reduce the number of IOs required for a full table scan.

   B. If we only have one set of data pages for a specific table, we can only have one index on that table.

   C. Bulkloading is used to reduce the number of IOs required to construct a B+ tree initially.

   D. Indices can help us reduce the number of IOs required for an UPDATE operation.

   E. The purpose of a fillFactor when bulkloading is to reduce the number of IOs required during the bulkLoading process.

Solution: C, D

A. False - they are for lookup on search key

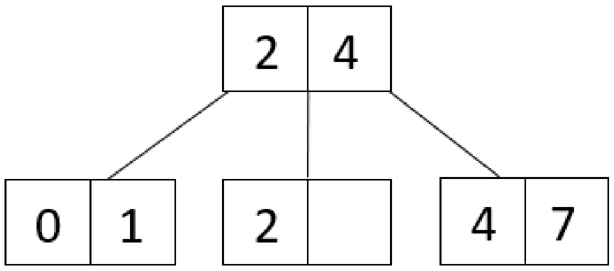
B. False - you can have many unclustered B+ trees on these datapages and one clustered.

C. True - that is the purpose of bulkloading.

D. True - takes less time to find the page (see discussion)

E. False - fillFactor prevents us from splitting as soon as we get new inserts.

Given the following order 1 B+ tree:



We now insert **5** into the tree. Answer questions 2 and 3 about the resulting tree.

2. (1 point) What keys are now in the root node? Separate them by comma if there are multiple, ordered from least to greatest.

Solution: 4

3. (1 point) What keys are now in the rightmost leafnode? Separate them by comma if there are multiple, ordered from least to greatest.

Solution: 5, 7

For the following questions we will be considering the Students table which has the columns: SID, GPA, and Units. Assume that we have an alternative 2 unclustered B+ tree of height 2 on the composite key <GPA, Units>. The table has 150 data pages in total and none of the index pages are in the buffer pool at the start of each part. Remember that we define the height of the root as 0 in this course.

4. (2 points) How many I/Os will it take to run the following query in the **worst case**:
`SELECT * FROM Students WHERE GPA = 4.0;`
Assume that **2** students have a GPA of 4.0 and they occur on the **same leaf node**.

Solution: 5 I/Os
3 I/Os to reach leaf + 1 to load first data page + 1 to load second data page = 5

5. (2 points) How many I/Os will it take to run the following query in the **worst case**:
`SELECT * FROM Students where Units = 16;`
Assume that **30** students have 16 units and they occur on **3 different leaf nodes**.

Solution: 150 I/Os
We cant use our index because we do not include the GPA as part of the query so we have to do a full scan.

6. (2 points) How many I/Os will it take to run the following query in the **worst case**:
`DELETE FROM Students where GPA=3.0 and Units = 18;`
Assume that exactly **1** student has a 3.0 GPA and has taken 18 units. Also assume that remove is implemented as it was on homework 2 (no rebalancing).

Solution: 6 I/Os
3 to reach leaf + 1 to read data page + 1 to write datapage + 1 to write leaf with removed key = 6 I/Os

5 Buffer Management (13 points)

1. (6 points) Which of the following statements are true? **There may be zero, one, or more than one correct answer.**
    A. After a request is finished, the requester of the page must set the dirty bit if the page was modified.
    B. A page in a pool cannot be requested multiple times.
    C. Clock policy is a good approximation for MRU
    D. Evicting a page from the buffer pool where the pin count is nonzero may cause unintended behavior.
    E. If a page has been accessed 5 times, the pin count can be anywhere between 0-5.
    F. When performing LRU with 4 buffer pages, the last 4 distinct elements in the access pattern will be in the buffer pool

Solution: A, D, E, F

B. A page in a pool can be requested multiple times (will have multiple pins on it then)

C. Clock policy is a good approximation for LRU

In the remaining questions, we are given an initially empty buffer pool with **4** buffer frames. Consider the following access pattern:

A C D B E E C A B D

In the next two questions, you will need to evaluate the clock policy, keeping track of three things: the pages in the buffer pool, the reference bits on the frames, and the number of buffer pool hits. Assume that the 4 frames are numbered 1 through 4, that the clock begins pointing to 1, and rotates through the frames in increasing order.
Assume that we do not advance the clock hand on a hit – i.e., when we request a page that is already in the buffer pool. We only advance the clock hand on a miss, as part of a page replacement. Be careful to follow this protocol!

2. (2 points) Using the CLOCK replacement policy, which frames have the reference bit set after the policy finishes? Fill in the appropriate boxes.

Solution: 2,3

See question below for explanation.

3. (2 points) Using the CLOCK replacement policy which pages remain in the buffer pool? Fill in the appropriate boxes.

Solution: ABDE

Point refers to the current frame it is on and count is the number of hits The clock policy would be run as follows: Char: A Clock: (A,1) (,0) (,0) (,0) point: 1 count: 0

Char: C Clock: (A,1) (C,1) (,0) (,0) point: 2 count: 0

Char: D Clock: (A,1) (C,1) (D,1) (,0) point: 3 count: 0

Char: B Clock: (A,1) (C,1) (D,1) (B,1) point: 0 count: 0

Char: E Clock: (E,1) (C,0) (D,0) (B,0) point: 1 count: 0

Match found!

Char: E Clock: (E,1) (C,0) (D,0) (B,0) point: 1 count: 1

Match found!

Char: C Clock: (E,1) (C,1) (D,0) (B,0) point: 1 count: 2

Char: A Clock: (E,1) (C,0) (A,1) (B,0) point: 3 count: 2

Match found!

Char: B Clock: (E,1) (C,0) (A,1) (B,1) point: 3 count: 3

Char: D Clock: (E,0) (D,1) (A,1) (B,0) point: 2 count: 3

4. (1 point) Start again with an initially empty buffer pool with **4** buffer frames. Consider the following access pattern:

   A B C D E A B C D E A B C D E

   Which of the following replacement policies is best suited for this access pattern? Select all of the above if the performance isnt affected by the replacement policy?

   A. MRU
   B. LRU
   C. Clock policy
   D. All of the above

Solution: A. MRU

This is a case that would be prone to sequential flooding. When there is sequential flooding, LRU/Clock do not perform well.

5. (1 point) What is the number of hits if MRU is used?

Solution: 8

6. (1 point) Now suppose we have **5** buffer frames instead and are starting with an initially empty buffer pool again. For the same access pattern in question 4, which of the following replacement policies is best suited for this access pattern? Select all of the above if the performance isnt affected by the replacement policy.

   A. MRU

B. LRU

C. Clock policy

D. All of the above

Solution: D. All of the above

We only access 5 pages and we have 5 buffer frames. This means that nothing ever has to be evicted so all the replacement policies will be the same

This page has been added for scratch work space.