CS532 Homework #4
Solutions
Max Total: 3 points

Name : Ankita Patra
B-Number : B01101280

Before you begin, for academic honesty, please read the paragraph below and sign there.
I have done this assignment completely on my own. I have neither copied nor shared my solution with anyone else. I acknowledge that if I engage in plagiarism or cheating, I will sign an official form admitting to the violation, which will be added to my official university record. I also understand that a first offense will result in a grade of 0 for the assignment and a one-level reduction in my course letter grade, and any subsequent offense of any kind will result in a grade of 'F' for the course

Name:____Ankita Patra_____
Signature:_____Ankita Patra_____

1. [1.5 points] Consider the following SQL query for a banking database system. For simplicity, assume that there is only one account type, and the status of a customer is gold, silver, or bronze.

```
select account_id, customer_name
from accounts
where status = 'gold' and customer_city = ‘Boston';
```

Assume the following for indexing:

• Each account tuple occupies 200 bytes (=196-byte data + a 4-byte pointer per tuple).

• The size of a page is of 2KB (=2000 bytes).

• There are 95 customers (out of total 10,000 customers) with gold member status.

• 73 customers live in Boston.

Given that, answer the questions below.

a) [0.5 point] If there is a secondary index on status and a secondary index on customer_city, which index the system must use first to optimize the query? Simply say ‘status’ or ‘customer_city’ [0.1point]. Briefly explain your answer [0.4 point].

Ans :
————-

The **customer_city** index should be the system must use first to optimizing the query.**(ans)**

Explanation:
————————-

-> When both status and customer_city are indexed as secondary (non-clustered) indexes, the optimizer should choose to use the one that returns fewer candidate tuples—> because fewer rows need to be checked further for the remaining condition.

In this case:

- The secondary index on **status** returns = 95 tuples.
- The secondary index on **customer_city** returns = 73 tuples.

-> Since 73 is less than 95.

-> The optimizer should first use the customer_city index.

-> This strategy minimizes the number of tuples that must be further filtered for the condition **status = ‘gold'** (according to the the indexing principles and selectivity) .

b)

[1 point] If there is a primary index on status and a secondary index on customer_city, the system will first use the primary index to find the gold customers and then select the gold customers in Boston. For simplicity,

assume that the entire B+ tree is in memory. Given that, answer the questions below:

Now we can assume the system has a **primary index on status** (which means the data is physically clustered by status) and a **secondary index on customer_city**.

Here is the few explanation for excitation :

Use the primary index on status to quickly retrieve all rows with **status = ‘gold'**.

Then, among these, use the secondary index (or simply filter) for **customer_city = ‘Boston'**.

b1) How many I/O pages the system needs to do [0.3 point]? Briefly explain your answer [0.5 point].

Ans :

We have below details :

Each account tuple is 200 bytes.

A page is 2000 bytes.

Number of tuples per page = 2000 / 200 = 10 tuples per page.

For 95 gold customers:

Pages needed = $\lceil 95 / 10 \rceil$ = 10 pages.(ans)

Therefor, The system requires **10 I/O pages** to fetch the gold customer records.

Brief explanation:

-> Because the primary index is clustered on status, all gold customer records are stored close together. With 95 gold customers and 10 rows per page, the gold records span 10 pages.

-> This calculation is consistent with the data-storage and indexing principles.

b2) Are they sequential or random I/O [0.1 point]? Briefly explain your answer [0.1 point]

Ans :

The I/O is **sequential**, since the gold records are clustered together due to the primary index on status.

Explanation:

When data is clustered (as with a primary index on status), the tuples are physically stored together. Thus, once the first relevant page is accessed, the remainder of the pages can be read one after the other in order. This results in sequential I/O, which is more efficient compared to random I/O.

2. [1 point] Using the nested loop join method, we want to process the join R ⋈ R.A = S.B S between two tables R and S. Assume that table R has N = 5,000 pages and table S has M = 100 pages, and the in-memory buffer has 110 pages in total. Assign in-memory buffer pages to R and S to minimize the number of I/O pages. Given that, answer the following questions.

(a) [0.5 point] How many I/O pages are necessary to execute this join? Briefly and clearly show your calculation. (A single line answer is expected.)

Ans :

I/O Pages Calculation:

-> Pages read for outer (S): 100

-> Since S fits in memory as one block, the inner relation (R) is scanned exactly once: 5,000 pages

-> **Total I/O pages = 100 + 5,000 = 5,100 pages**

(b) [0.5 point] How many I/O operations are necessary to execute this join? Briefly and clearly show your calculation. (A single line answer is expected.)

Ans :

I/O Operations Calculation:

- **1 I/O operation** to load **S** into memory.
- **500 I/O operations** to scan **R** (since we can load **10 pages** of **R** at a time).
- No extra I/O is required to access **S** because it's already in memory.
- Total I/O operations= 1+ ceil(N/(K-M)) = 1+500 = 501

Thus, the total **I/O operations** needed to execute this join is **501**.

3. [0.5 point] For three tables R1(A, B, C), R2(C, D, E), and R3(E, F), assume the

following:

• R1.C is a non-null foreign key referencing R2.C.

• R2.E is a non-null foreign key referencing R3.E.

• R1 has 2000 tuples, R2 has 1500 tuples, and R3 has 3000 tuples.

To join the three tables together, we consider two query executions below:

• Plan A: (R1 ⋈ R2) ⋈ R3

• Plan B: R1 ⋈ (R2 ⋈ R3)

Given all the information above, answer the following questions:

a) [0.1 point] Which plan will be faster? Simply say Plan A or B. No explanation is needed.

Given details according to the question :

- **Tables and Tuple Counts:**
  - R1(A,B,C): 2000 tuples
  - R2(C,D,E): 1500 tuples
  - R3(E,F): 3000 tuples
- **Foreign Key Relationships:**
  - R1.C is a non-null foreign key referencing R2.C
    → Each R1 tuple has exactly one matching tuple in R2.
  - R2.E is a non-null foreign key referencing R3.E
    → Each R2 tuple has exactly one matching tuple in R3.

Faster plan: **Plan A (ans)**

b) [0.2 point] How many tuples in total are produced by Plan A including intermediate tuples produced during the query processing [0.1 point]? Briefly explain your answer [0.35 point].

Ans :

Plan A: (R1 ⋈ R2) ⋈ R3

1. **First Join (R1 ⋈ R2):**
   Since every tuple in R1 (2000 tuples) finds exactly one matching tuple in R2 (by the foreign key constraint), the result produces **2000 tuples**.

2. **Second Join ((R1 ⋈ R2) ⋈ R3):**
   In the join between the intermediate result (with 2000 tuples from R1 ⋈ R2) and R3 (via R2.E = R3.E), every tuple in the intermediate result finds exactly one matching tuple in R3.
   Therefore, the final join produces **2000 tuples**.

3. **Total Intermediate Tuple Count:**
   Here the intermediate results are:

   - From R1 ⋈ R2: 2000 tuples
   - Final join result: 2000 tuples
     Total = 2000 + 2000 = 4000 tuples

Therefore , Plan A produces a total of 4000 intermediate tuples (2000 from R1 ⋈ R2 plus 2000 from joining with R3).

Brief Explanation: The first join yields 2000 tuples (one per R1 tuple) and joining these with R3 produces another 2000 tuples, totaling 4000 tuples across the intermediate results.

c) [0.2 point] How many tuples in total are produced by Plan B including intermediate tuples produced during the query processing [0.1 point]? Briefly explain your answer [0.35 point].

Plan B: R1 ⋈ (R2 ⋈ R3)

1. **First Join (R2 ⋈ R3):**
   Since every tuple in R2 (1500 tuples) has exactly one matching tuple in R3 (by the foreign key constraint), the intermediate result is **1500 tuples**.

2. **Second Join (R1 ⋈ (R2 ⋈ R3)):**
   Now R1 (2000 tuples) is joined with the R2 ⋈ R3 result on the condition R1.C = R2.C. Since every R1 tuple has a matching tuple in R2 (and thus in the intermediate join), the final join produces **2000 tuples**.

3. **Total Intermediate Tuple Count:**

   - From R2 ⋈ R3: 1500 tuples

- Final join result: 2000 tuples
  Total = 1500 + 2000 = 3500 tuples (ans)

Explanation: The first join produces 1500 tuples (one per R2 tuple) and joining these with R1 produces 2000 tuples, for a **total of 1500 + 2000 = 3500** tuples across the intermediate results.