CS 348 - Homework 4: Functional Dependencies, Normalization, and Indexes.

(100 Points)

Fall 2020
Due on: **11/06/2020 at 11:59 pm**
This assignment is to be completed by individuals. You should only talk to the instructor, and the TA about this assignment. You may also post questions (and not answers) to Campuswire.

There will be a 10% penalty if the homework is submitted 24 hours after the due date, a 15% penalty if the homework is submitted 48 hours after the due date, or a 20% penalty if the homework is submitted 72 hours after the due date. The homework will not be accepted after 72 hours, as a solution will be posted by then.

Submission Instruction: Write your answers on this word file and generate a pdf file. **Upload the pdf file to Gradescope**.

Question 1:
Consider the following relation about blood-sugar readings:

BSR(PatientID, Date, PatientName, PatientAge, SugarLevel, InsulinDose)
The BSR relation has the following functional dependencies:
PatientID, Date -> SugarLevel
PatientID -> PatientName, PatientAge
SugarLevel -> InsulinDose.

To get an idea of the domain of sugarLevel and insulinDose attributes, you can look at the following table:
https://www.researchgate.net/figure/The-recommended-insulin-dosage-according-to-the-patients-glucose-level-Blood-sugar_tbl1_262295520

a. Give an example instance where the function dependency SugarLevel -> InsulinDose is violated (include only two or three rows). (5 points)

   Solution:

PatientID	Date	PatientName	PatientAge	SugarLevel	InsulinDose
1	10/22/20	Alice	29	140	2
1	10/23/20	Alice	29	140	3
2	10/23/20	Bob	42	199	3

The first two records violate SugarLevel -> InsulineDose because for the same SugarLevel we have two different InsulinDoes values.

b. In what normal form is the above relation? (5 points)

Solution: The relation is not in 3NF since there are transitive dependencies with InsulinDose. The relation is not in 2NF because there are partial dependencies since PatientName and PatientAge depend only on PatientID and not the complete key. Hence, the relation is in 1NF.

c. Decompose the BSR relation into BCNF. **In each step of your decomposition, show the FDs you are lifting and the resulting tables.** (5 points)

Solution:
We lift PatientID -> PatientName, PatientAge *and breakdown BSR into two tables:* PatientBSR *and* Patients.

PatientBSR(PatientID, Date, SugarLevel, InsulinDose)
PatientID, Date -> SugarLevel
SugarLevel -> InsulinDose

Patients(PatientID, PatientName, PatientAge)
PatientID -> PatientName, PatientAge

We further breakdown PatientBSR by lifting SugarLevel -> InsulinDose *and further breakdown* PatientBSR *into two tables:* PatientBSR *and* SR.

PatientBSR(PatientID, Date, SugarLevel)
PatientID, Date -> SugarLevel

SR(SugarLevel, InsulinDose)
SugarLevel -> InsulinDose

Patients(PatientID, PatientName, PatientAge)
PatientID -> PatientName, PatientAge

The relations are already in BCNF since every left-hand side of each FD is a key (super key).

d. Show an example of a lossy decomposition of BSR. Explain briefly why your decomposition is lossy. (5 points)

Solution:

BSR1(PatientID, Date, PatientName, PatientAge)

BSR2(Date, SugarLevel, InsulinDose)

The above is a lossy decomposition since we cannot get BSR by joining the two tables (BSR1 and BSR2). The common attribute in the two relations, which is date, is not a key in either relation. Therefore, joining BSR1 and BSR2 using the date attribute will introduce new rows that were not originally in BSR.

Question 2:
Suppose the BSR relation in Question 1 has hundreds of thousands of rows – too many for a person to read. We want to know whether or not this table currently satisfies the functional dependency SugarLevel -> InsulinDose. This is critical data where checking such FD can be lifesaving. Explain how to use one or more SQL queries to test if this FD is satisfied in the relation. Make sure you explain how the results of your queries determine whether the FD is satisfied (e.g., if two queries return different number of rows then the FD is violated).

a. Give a solution using the aggregate function COUNT. (10 points)
   Solution:
```
SELECT COUNT(DISTINCT SugarLevel) FROM BSR;
SELECT COUNT(DISTINCT SugarLevel ,InsulinDose) FROM BSR;
```

   If the two queries return different results, then the FD is violated

b. Give a solution using GROUP BY and HAVING. (10 points)
   Solution:
```
SELECT SugarLevel FROM BSR
GROUP BY SugarLevel
HAVING COUNT(DISTINCT(InsulinDose) > 1;
```

   This should return an empty set if the FD holds. If any rows are returned then the FD is violated.

c. Give a solution that joins the BSR relation to itself (no GROUP BY, HAVING, and aggregate functions). (10 points)
   Solution:
```
SELECT * FROM BSR b1, BSR b2
WHERE b1.SugarLevel=b2.SugarLevel
AND b1.InsulinDose <> b2.InsulinDose;
```

   This should return an empty result if the FD holds. If any rows are returned then the FD is violated.

Question 3:

Assume we have a table that includes student information:

```
Student(ID, username, name, Age, Height)
     Where ID is a key, and username is a key
```

There are four indexes on this table:

I1: Unclustered hash index on <ID>

I2: Unclustered B+ tree index on <username>

I3: Clustered B+ tree index on <name>

I4: Unclustered B+ tree index on <Age>

I5: Unclustered B+ tree index on <Height>

For each of the WHERE conditions below, say which index you think is best to use and **why** you think it's the best. **Choose only one index.**

a. (5 points):

```
Select * From Student
Where username="theLuckyOne" AND name Like 'A%';
```

Solution: **Unclustered B+ tree index on <username>**.

Explanation: username is a key in the table (unique). Therefore, using the I2 index on username, we need to read one index leaf page then follow the pointer (the RID for the record with username 'theLuckyOne') to read the data page containing the record. The record can then be checked to see if the name starts with the letter 'A'. Even though I3 is clustered on name, the condition on name is not selective (likely, we have a large number of students whose name starts with the letter A). Therefore, using I3, we need to read a large number of leaf and data pages.

b. (8 points):

```
Select * From Student
Where (name >= 'a' AND name < 'b')
       AND (username >= 'a' AND username < 'b');
```

Assume that the number of students having a name that starts with the letter ‘a’ is the same as the number of students with a username that starts with the letter ‘a’.

Solution: **Clustered B+ tree index on <name>**

Explanation: The I3 index is clustered. Therefore, the data records are sorted on name. Using I3, we have fewer data pages to access. The records with a name that starts with the letter ‘a’ are adjacent to each other in the data file. Therefore, those records are located in relatively fewer pages. However, since the data file is sorted on name, records with a username that starts with the letter ‘a’ are likely to be scattered in the data file (the worst case is having the 20 records located in 20 data pages).

Once the records with the specified name range are retrieved using I3, we can check the username criteria and keep only data records where username starts with the letter ‘a’.

c. (10 points):

```
Select * From Student
 Where (Age < 18 OR Age = 38 OR Age > 50)
       AND (Height > 6.5)
```

Assume the number of students having an age in the specified range is the same as the number of students whose height is larger than 6.5.

Solution: **I5 Unclustered B+ tree index on <Height>**

Traversing I5 is straightforward and will require less page IOs to read the index leaf pages. All leaf pages are on the right side of the index and are *adjacent* to each other. The search starts with locating the leaf page with the first height that is greater than 6.5. We call this leaf page *firstLeaf*. Other heights in the required range are either in firstLeaf or in the ones to the right of firstLeaf. We can use the linked list that connects the leaf pages for easy traversal.
As in the previous questions, after retrieving records with the required height range, we can check whether each record satisfies the age range.

Using the I4 index, on the other hand, may require scanning all leaf pages (by following the linked list from beginning of the list to the end). Consider the linked list that connects all leaf pages. The required leaf pages (that have data entries for the specified age range) include some in the beginning of the linked list (with age <18), some in the middle (age = 38), and some in the end (age > 50).
Another alternative, is to scan for each range separately (i.e., traversing the index for age <18, then restart traversal from the root for the other ranges). This approach is also slightly more expensive than using the height index, which requires traversing the root and other internal nodes only once.

Question 4:

It is sometimes possible to evaluate a particular query using only indexes, without accessing the actual data records. This method reduces the number of Page IOs and hence speed up the query execution.

Consider a database with two tables:

`Book(ISBN, title, year, publisher)`

`Sells(ISBN, vendor, price)`

Assume three unclustered indexes, where the leaf entries have the form [search-key value, RID] (i.e., alternative 2).

`I1: <year>` on `Book`
`I2: <publisher>` on `Book`
`I3: <price, ISBN>` on `Sells`

For the following queries, say which queries can be evaluated *with just data from these indexes*.

- If the query can, describe how by including a simple algorithm.
- If the query can't, briefly explain why.

a. (zero points, answer is included)
SELECT MIN(price)
FROM Sells;
Solution: The query can be evaluated from the `<price, ISBN>` index. The

query result is the price part of the search-key value of the leftmost leaf entry in the leftmost leaf page.

b. (5 points)
SELECT AVERAGE(price)
FROM Sells;
Solution: The query can be evaluated from the <price, ISBN> index. The result is the average of all prices in all leaf pages (we have to iterate over all search-key values in all leaf nodes and extract the price part)

c. (4 points)
SELECT AVERAGE(price)
FROM Sells
GROUP BY vendor;
Solution: The query cannot be evaluated with the data from the index <price, ISBN> because the search key has no information regarding vendors.

d. (8 points)
SELECT COUNT(*)
FROM Book
WHERE publisher = 'Knopf' AND year = 2010;
Solution: The query can be computed by using the <publisher> and <year> indexes. Use the <publisher> index to find RIDs for Knopf books. We call this set of RIDs S1. Use the <year> index to find RIDs for books published in 2010. We call this set of RIDs S2. Compute the intersection of S1 and S2 and store the result in S3 (S3 contains RIDs of records that are both published by Knopf and published in 2010). The final result is the number of elements in S3 (i.e., |S3|).

e. (10 points) (think carefully about this one!):
SELECT ISBN, AVERAGE(price)
FROM Sells

GROUP BY ISBN
HAVING COUNT(DISTINCT vendor) > 1;

Solution: The query can be evaluated from the `<price, ISBN>` index.

- Fetch all the leaf pages with `<price, ISBN>` and then sort the data entries by ISBN (in the index, data entries are sorted by price then ISBN).
- For each ISBN, compute the average price and the number of data entries the ISBN has.
- Return ISBN and average price for any ISBN whose number of entries is greater than one. Having multiple (>1) entries for an ISBN indicates that the ISBN has multiple vendors (because ISBN,vendor is a key). If we have a single data entry of an ISBN, we discard that ISBN.