Sample solutions for ISYS2120 Assignment 4 (sem1, 2024)

A [relational algebra] (1 point)
Note: if you have difficulty producing relational algebra symbols, you may use text, with SEL_{C} to represent σ_C and PROJ_{D} to represent π_D and X JOIN Y to represent X ⋈ Y.

Consider the following relational schema which stores information about the selling operations of a store.

```
CREATE TABLE Part (PId INTEGER,
                   PDesc VARCHAR(30),
                   PUsualPrice INTEGER NOT NULL,
                PRIMARY KEY (PId));
CREATE TABLE Customer (CId INTEGER,
                       CName VARCHAR(20),
                       CAddr VARCHAR(40),
                 PRIMARY KEY (CId));
CREATE TABLE Cust_Phone (CId INTEGER,
                         CPhone CHAR(15),
                         CPhoneFee INTEGER,
                PRIMARY KEY (CId, CPhone),
                FOREIGN KEY (CId) REFERENCES Customer(Cid));
CREATE TABLE Sale (SId INTEGER,
                   SDate DATE,
                   CId INTEGER NOT NULL,
                   PId INTEGER NOT NULL,
                   SPrice INTEGER NOT NULL,
                PRIMARY KEY (SId),
                FOREIGN KEY (CId) REFERENCES Customer(Cid),
                FOREIGN KEY (PId) REFERENCES Part(Pid));
```

In **Part** a row (31, 'Large Widget', 3050) represents that the part whose PId is 31, is described as "Large Widget" and the usual price is $30.50 (that is, PPrice is measured in cents). In **Customer**, a row (27, 'Jay Lui', '20 Pine St, Sydney') represents that the customer whose CId is 27 is named "Jay Lui", with address "20 Pine St, Sydney. In **Cust_Phone**, a row (27, '0414441777', 300) represents that the customer with CId 27 has a phone which can be reached as 0414662306 and the customer will pay a fee of $3 for each support call they make on that phone (CPhoneFee is measured in cents). In **Sale**, a row (110, '2022-01-20', 27, 31, 2990) represents that the sale whose SId is 110 occurred on 20 January 2022 when customer with CId 27 purchased the part with PId 31, for a price of $29.90 (SPrice is measured in cents).

Write *two different (but logically equivalent) relational algebra expressions* involving the tables in the schema above, so that each has a result which will answer the request ""Find the SId of every sale where the sale price is less than one half of the usual price of the part being sold"

Some of the possible answers:

- PROJ_{SId} (SEL_{SPrice < 0.5*PUsualPrice} (Part NATURALJOIN Sale))
- PROJ_{SId} (SEL_{SPrice < 0.5*PUsualPrice} (PROJ_{SId,SPrice,PUsualPrice}(Part NATURALJOIN Sale)))
- PROJ_{SId} (SEL_{SPrice < 0.5*PUsualPrice AND Part.PId = Sale.PId} (Part CROSS Sale))
- PROJ_{SId} (SEL_{Part.PId = Sale.PId} SEL_{SPrice < 0.5*PUsualPrice} (Part CROSS Sale))
- PROJ_{SId} (SEL_{SPrice < 0.5*PUsualPrice} ((PROJ_{PId,PUsualPrice}(Part)) NATURALJOIN (PROJ_{SId,PId,SPrice}(Sale)))
- PROJ_{Sid}(Sale) - PROJ_{SId} (SEL_{SPrice >= 0.5*PUsualPrice} (Part NATURALJOIN Sale))

In each of the above, you can also reverse the order of the tables such as Part, Sale in the natural join or cross-product

Advice to avoid errors: it is important not to project too soon (if a column is not available, because it was left out of a projection, it can't be used in a later selection)

B [relational design theory] (2 points)

Consider a schema which has one table the following relational design, which collects data about the air flights. There is a single table: Crew(flightCode,destination,date,tailCode,acName,airline,emp_id,name,title).

In **Crew**, a tuple ('QF141', 'Dubai', '2019-10-31', 'VH-EAB', 'City of Canberra', 'Qantas', 65098, 'Joe Bloggs', 'Navigator') indicates that on 31st October 2019, flight QF141 departed for Dubai on Qantas aircraft VH-EAB ("City of Canberra") with navigator Joe Bloggs (employee ID 65098) on board. Note that tailCode is the government issued identifier for an aircraft. The following functional dependencies are valid in this schema:

- **flightCode → destination**
- **flightCode, date → tailCode**
- **tailCode → acName, airline**
- **emp_id → name, title**

B(i) Explain in English the meaning of the fd flightCode, date → tailCode. Also, give an example of data that would not be allowed in the table **Crew** because of this dependency.

Some of the possible answers:

"For a given combination of flightCode and date, there is at most one tailCode", "It is not possible to have two different tailCodes for the same combination of flightCode and date"

Here is one example showing a state of the database that would not be allowed, but there are many many other examples!

('QF141', 'Dubai', '2019-10-31', 'VH-EAB', 'City of Canberra', 'Qantas', 65098, 'Joe Bloggs', 'Navigator')
('QF141', 'Dubai', '2019-10-31', 'PL-MLY', 'City of Darwin, 'Qantas', 62077, 'Sue Smith', 'Captain')

Advice to avoid errors: Give a full example that fits the schema, not just the columns that are relevant to the issue

B(ii) Based on this schema and its fds, calculate the attribute closure (flightcode, empid)+ Show the step-by-step working of the calculation.

Here is one solution, but we can apply the fds in different order, giving different intermediate steps.

Initially, result = {flightCode, empid}

Use flightCode -> destination, as flightCode is a subset of result, we add destination to result.

Now result = {flightCode, destination, empid}

Use empid -> name, title, as empid is a subset of result, we add name and title to result.

Now result ={flightCode, destination, empid, name, title}.

As neither so-far-unused fd has lhs a subset of result, we can not expan result more.

So (flightCode, empId)+ = (flightCode, destination, empid, name, title).

Advice to avoid errors: it is important to give a reason why the calculation steps have finished (and also, it is not valid simply to note that each fd has been checked already; we need that no unused fd applies to the final result, so each must be checked after the last expansion of result). Also, it is essential to include the initial columns in the closure.

B(iii) State whether the relation **Crew** is in BCNF. Justify your answer.

A sample answer, but one could use other fds in the same way, instead of empid -> name, title as I do below.

Consider the fd empid -> name, title. This is not trivial (because rhs is not a subset of lhs), and lhs is not a superkey of Crew relation, because we can calculate empid+ as (empid, name, title) is is not all columns of Crew. So this fd is non-trivial and its lhs is not a superkey; this shows that Crew violates the BCNF property.

Advice to avoid errors: Give a justification for saying that the lhs is not a superkey of Crew; mention that the chosen fd is not trivial

B(iv) *Give a lossless-join decomposition* of **Crew** into **two** relations. For each decomposed relation, *state the functional dependencies* that hold for it, and *state a primary key* for it. *Justify* that your decomposition has lossless-join property.

Sample answer, but there are several others, as one can use each of the fds to do the decomposition.

We decompose Crew using the fd empid -> name, title.

We then get two relations, R1(empid, name, title) and R2(flightCode,destination,date,tailCode,acName,airline,emp_id).

For R1(empid, name, title), the fd is empid -> name, title, and a primary key for R1 is empid.

For R2, the fds are flightCode -> destination; flightCode, date -> tailCode; tailCode -> acName, airline. A primary key for R2 is (flightCode, date, empid).

This is a decomposition because all the columns of Crew occur in R1 or R2, and there is some overlap between the columns of the two new relations. It has lossless-join property because the common column empid is a superkey of R1.

Advice to avoid errors: make sure you give a decomposition (include all columns of Crew in the new relations, make sure there is some overlap between the two new relations).

C. [index choices] (1 point)

Consider again the relational schema shown in Question A, and the Cust_Phone table. Suppose that no CREATE INDEX commands have been executed; that means that the structure of the table is as a tree-based clustering index on the primary key (CId, CPhone).

Consider the SQL query below, that refers to the sales schema shown on page 2.

```
SELECT CId
FROM Cust_Phone
WHERE CPhone = '0414441777' AND CPhoneFee > 500
```

C(i) Explain why this query cannot be answered effectively using the primary index, and state how the query would be answered with the initial structure of the table.

Sample answer. The primary index is arranged by CId, and within Cid by CPhone. As the query does not put any restriction on the value of CId, the index can't be used to find relevant records (they could be pointed at from anywhere in the index). So the query will instead do a table scan, looking at each data record and flitering to see whether CPhone and CPhoneFee meet the WHERE clause; if so, CId will be output.

Advice to avoid errors: it is important to mention that the query doesn't give information about the prefix of the index (it is not sufficient to discuss that the query does involve columns that are not in the index)

C(ii) Write SQL to create an index that will allow much faster execution for the query above, when the database has a large amount of data in the table. Also, explain how the index would allow the query to be calculated.

Here are several possible answers, in decreasing value.

Possibly the most speedup would be from a covering index on (CPhone, CPhoneFee, CId), by
CREATE INDEX IxA on Cust_Phone(CPhone, CPhoneFee, CId)
This could be used to answer the query by descending the index to the first entry after ('0414441777', 500, *). All the immediately following index entries will be taken, and the CId returned, until we get to the first one following where CPhone is no longer 0414441777.

Not quite as fast would be CREATE INDEX IxB on Cust_Phone(CPhone, CPhoneFee). This could be used by by descending the index to the first entry after ('0414441777', 500). All the immediately following index entries will be looke dup in the data, and the CId returned from there, until we get to the first one following where CPhone is no longer 0414441777.

Somewhat slower, but still useful: CREATE INDEX IxC on Cust_Phone(CPhone). This could be used by descending the index to the first entry with ('0414441777'). All the immediately following index entries will be looked up in the data, and CPhoneFee checked; if it is greater than 500, the CId will be returned from that data record, until we get to the first one following where CPhone is no longer 0414441777.

Advice to avoid mistakes: In cases where the index is composite, its important to mention where in the index the relevant entries are found (especially, if they arte immediately following one another in the index, that is significant, because they will be accessed together). When some aspect of the WHERE clause can be checked in the index, one should not say it will be checked again as a filter on the data records.

D. [Database-backed web-app security] (1 point)

In the code you used as a skeleton for asst3 (and which had previously been used in lab for week 8), access to the data in the database is given only to certain database accounts (the data owner, and any account to which the data owner

granted permissions). In the following questions refer to the webapp as given in the skeleton, without any of your modifications.

D(i) Describe how the password for a database account is provided to the dbms, and what security protects this password.

A sample answer (maybe a bit more detailed than needed, though not covering everything)
The password for the database account (as well as the name of that account, the location of the database server, the name of the database, etc) are held in a file on the app-server machine, the file is called config.ini. On every request, the web app reads that password in the database_connect() method, using the read() method of configparser.ConfigParser(), and then stores it within the connectiontarget variable (a dictionary), and then actually tries to connect using pg8000.connect() with the password as a parameter. The pg8000 method sends messages back and forth with the postgresql server; the details depend on how the postgresql server has been set up; if the server is done well [as happens with the webapp code and the uni postgresql server we told you to use, which then use scram-sha-256 as the protocol], then the messages do not contain the password itself, but a hashed version of a combination of the password with salt, and also an encoding of once-only values provided by each party. Whether those messages are also encrypted before being sent (eg sent with SSL), is also decided between the dbms server and the app. With the postgresql server used in lab, and the webapp code we provided, SSL is not used, and the messages are clear on the network. The postgresql server at uni is also set up to only be take messages sent from the uni network or VPN.

This provides fairly good protection against complete outsiders who don't have access to the file system where the config file is stored, and the sha-256 approach ensures that the password is not visible to someone intercepting the network, and also the messages are not usable if being replayed later (and the network being inside the uni also keeps some attackers away).

Advice to avoid errors: This is not asking about the end-user accounts managed by the app; it is about the database accounts. It is also about the specific code from lab8, not general issues with webapp coding.

D(ii) *Identify any weakness* in the security this provides, and *discuss* how this security mechanism could be improved.

Sample answer (again perhaps more detailed than needed): The password is stored directly in config.ini, and the zip we provided of the scaffold, will install this as openly readable to any account on the file system. This can be improved by, at the least, making the file readable only by the owner (and sysadmins of course). Other possible improvements could be encrypting the file, with the encryption key kept elsewhere; or using OS environment variables rather than a file for configuration information.
Another potential weakness is because the webapp doesn't enforce a strong authentication protocol, nor does it enforce use of SSL, but these depend on how

the postgresql server was set up. So the same webapp could be used with a different dbms server, and the password might be visible on an attackable network; changing the webapp to insist on SSL would be wise.

Advice to avoid errors: Do not suggest requiring frequent changes of password; this is generally considered to make security worse, as people often choose simpler passwords if they need to invent them often.