Solution sketches for the 24-hour Data Science take home exam, 2020

Disclaimer: In the following, we present solution sketches for the various questions in the exam. These solution sketches are provided only as a reference, and may lack details that we would expect in a complete answer to the exam. Moreover, some of the questions may admit more than one correct solution, but even in such cases only one solution sketch is provided for brevity.

Note, in addition, that the evaluation of the exam takes into account our expectations regarding solutions, the actual formulations provided, the weights of various questions, but also and most importantly the overall evaluation of the exam assignment as a whole. This evaluation is performed by internal examiners and grades are finally provided by discussion and consensus. As such, it is not advised to reason about final grades based on this document.

Final Exam, Data Science, 2020

This is the individual 24-hour take-home portion of the exam of Development of Information Systems (UIS) / Data Science (DS). The exam is made available via Digital Exam on June 19, 2020, 9:00 and your solution should be handed in via Digital Exam by June 20, 2020, 9:00. To solve the exam, fill in the form elements for all the questions directly in this PDF file. In one of the questions, you are asked to create two drawings. Please create these as a separate PDF file (e.g. by taking a picture of a hand-drawing), and then add them as extra files (bilag) in digital exam when submitting. Please limit yourself to only PDF format for such attachments.

Hand-ins for this exam must be individual. Cooperation on or discussion of the contents of the exam with other students is strictly forbidden. The solution you provide should reflect your knowledge of the material alone. The exam is open-book and you are allowed to make use of the book and other reading material of the course. If you use on-line sources for any of your solutions, they must be cited appropriately. By submitting a solution, you commit to have abided by the academic integrity expectations at the University of Copenhagen.

Question 1: Queries (30%)

A group of your friends are creating a website, `compatiblebabystuff.com`, that lists baby products (e.g., car seats, pacifiers, strollers, among others) along with the baby activities in which they can be safely used (e.g., traveling, deep sleeping, playing, among others). The designers of `compatiblebabystuff.com` have enlisted you to compose a set of queries for their web application and for data analysis activities. They have designed the following relational database schema to record the information for the website:

```
baby_product(pid, name, type)
baby_activity(aid, description)
compatibility(pid, aid, level)
```

The `baby_product` relation records each product with their ID, name (e.g., 'ABC Toys Pacifier', and type (e.g., 'pacifier'). The `baby_activity` relation includes the activity ID and its description (e.g., 'deep sleeping'). The `compatibility` relation records which products are compatible with which activity and at what level. In detail, the `level` attribute in this relation can take the values 'compatible', 'incompatible', or 'supervised'. For example, a car seat can be compatible with traveling (level 'compatible'), but be incompatible with deep sleeping (`level` 'incompatible'). An infant support pillow can be used for playing if the baby is supervised by an adult (`level` 'supervised'). Note that in the relations above, underlines denote primary keys while italics denote foreign keys.

Answer each of the queries below in the language requested. *NOTE: If the query is formulated in a language different than the one requested, the answer will be disconsidered. If the language is Relational Algebra or Extended Relational Algebra, you should use in your answer LaTeX notation as in the `relational_algebra_cheatsheet.pdf` document in Absalon under the `sql` folder in Files.*

(a) List pairs of product IDs that are both compatible with the activity 'deep sleeping' (`level` 'compatible'). *NOTE: Each pair of IDs should include different products and be output only once, i.e., pairs (i, i) should not be included and if pair (i,j) is output, then pair (j,i) should not.* [Language: Relational algebra]

```
P1  := \pi_{pid}(\sigma_{level = 'compatible'}(compatibility)
              \natjoin
              \sigma_{description = 'deep sleeping'}(
                  baby_activity))

P2  := P1

Res := P1 \thetajoin_{P1.pid < P2.pid} P2
```

(b) List the types of baby products and for each type, the total number of activities with which the products of that type are compatible or requiring supervision (`level` 'compatible' or `level` 'supervised') and the total number of activities with which the products of that type are incompatible (`level` 'incompatible') [Language: Extended relational algebra]

```
CS_prod := \gamma_{type, COUNT(*)CS_total}(
              baby_product
              \natjoin
              \sigma_{
                level = 'compatible'
                \vee level = 'supervised'}(
               compatibility))
I_prod  := \gamma_{type, COUNT(*)I_total}(
              baby_product
              \natjoin
              \sigma_{
                level = 'incompatible'}(
               compatibility))
Res     := CS_prod \natjoin I_prod
```

(c) List the combinations of product ID and activity ID that have not been recorded in the `compatibility` relation. [Language: SQL]

```
SELECT p.pid, a.aid
FROM   baby_product p, baby_activity a
EXCEPT
SELECT pid, aid
FROM   compatibility
```

(d) List the descriptions of the activities that deemed compatible (`level` 'compatible') with a product of type 'activity gym', but incompatible (`level` 'incompatible') with a product of type 'changing table'. *NOTE: In the set of activities selected, you should include exactly the same number of duplicate activity descriptions as originally in the* ***`baby_activity`*** *relation (not more, not less).* [Language: SQL]

```
SELECT a.description
FROM   baby_activity a
WHERE  EXISTS (SELECT *
               FROM   compatibility c1 INNER JOIN
                      baby_product b1  USING (pid)
               WHERE  c1.aid = a.aid
                 AND  c1.level = 'compatible'
                 AND  b1.type = 'activity gym')
  AND  EXISTS (SELECT *
               FROM   compatibility c2 INNER JOIN
                      baby_product b2  USING (pid)
               WHERE  c2.aid = a.aid
                 AND  c2.level = 'incompatible'
                 AND  b2.type = 'changing table')
```

Question 2: Indexing (15%)

Consider the B+ Tree in Figure 1:

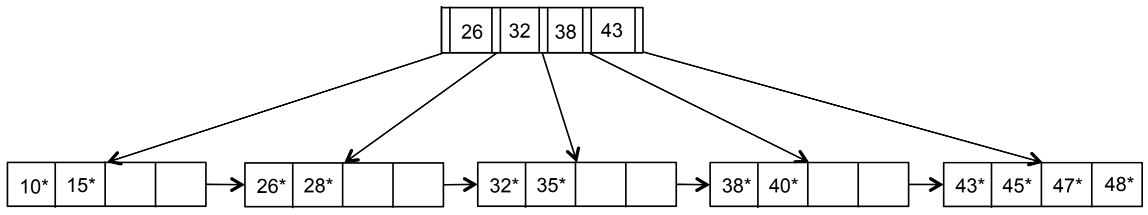


Figure 1: B+ Tree for Question 2. NOTE: We use the notation in the slides of the lecture on "Indexing". You may assume here that "*" denotes a pointer to the record associated with the key.

Now, answer the following questions:

(a) Starting from the tree in Figure 1, show the final resulting tree after the insertion of keys 50*, 51*, and 52* along with a short justification documenting which key insertions caused splits of nodes containing data entries or index entries and which did not.
Write justification here - add image of tree (with label "2(a)") as PDF merged in at the end of this file.

The insertion of 50* causes a split at the rightmost leaf, and an insertion of separator 47 is cascaded upwards to the root. This in turn causes another split where separator 38 is promoted as a new root. The insertion of 51* is accommodated in the new rightmost leaf, filling it up (47*,48*,50*,51*). The insertion of 52* then causes a split in this rightmost leaf, cascading an insertion of separator 50, which is accommodated in the parent node (43, 47, 50).

(see appendix at end of file)

(b) Starting again from the tree in Figure 1, show the final resulting tree after the deletion of keys 10*, 15*, and 26* along with a short justification documenting which key deletions caused merges of nodes containing data entries or index entries and which did not.
Write justification here - add image of tree (with label "2(b)") as PDF merged in at the end of this file.

The deletion of 10* causes an underfull leftmost leaf, which is merged with its sibling. This causes the deletion of separator 26 at the root. The deletion of 15* is absorbed in the new leftmost leaf, but the further deletion of 26* causes this leaf to be underfull and merged with its sibling, resulting in a new leftmost leaf (28 *,32*, 35*). The merge causes the deletion of separator 32 from the root, which absorbs that deletion without need for any further merges.

(see appendix at end of file)

(c) Assume the tree above is a nonclustered index over the age attribute of a relation `students(sid, name, age)`, where the relation itself is represented as a heap file. Consider a query that retrieves the names of all student with ages between 30 and 40. What steps will the DBMS need to take to execute this query? How many pages would you expect the DBMS to bring from disk to memory? Why? (*NOTE: You should assume that when the query processing starts, no index or heap file pages are already in memory*)

The query processing starts by locating key 30 in the B+ tree. This accesses the root plus one leaf node and finds that 30 is not there. A range scan is then started at the leaf level stopping at keys <=40, accessing another two leaf nodes. So that correspond to 4 index I/Os. Since the index is nonclustered, each key scanned within the range generates in principle one extra I/O to the heap file. That is 4 keys, corresponding to 4 heap file I/Os. So 8 I/Os are carried out in total.

Question 3: Database Design and Triggers (30%)

With the rise of Internet of Things (IoT) applications, a team of engineers sets up a service called `thriftyiot.com` wherein they offer pay-as-you-go sensor measurements for a variety of sensor types. To record the sensor data for their customers, the engineers of `thriftyiot.com` designed the following relational database schema:

```
sensor_measurement(sensor_id, m_timestamp, sensor_type,
                   value, marginal_cost)
sensor_type(sensor_type, marginal_cost)
sensor(sensor_id, sensor_type)
```

The `sensor_measurement` relation records the values of measurements of sensors over time, while the `sensor_type` relation records the marginal (monetary) cost of making an extra measurement with a sensor of the given type. Finally, the `sensor` relation records the existing sensors and their types. Note that in the relations above, underlines denote primary keys while italics denote foreign keys. The following non-trivial functional dependencies apply over the attributes of the relations:

```
sensor_id → sensor_type
sensor_id, m_timestamp → value
sensor_type → marginal_cost
```

Since the billing system of `thriftyiot.com` works online and under low latency demands, the engineers of `thriftyiot.com` have decided to keep the `sensor_type` and the `marginal_cost` also directly into the `sensor_measurement` relation, even though this decision introduces some data redundancy.

Based on this scenario, answer the following questions about design theory:

(a) What are all the keys of `sensor_measurement`? Why?

We observe that the value attribute is only determined by the combination of sensor_id and m_timestamp and that sensor_id and m_timestamp are not determined by any other attributes. So any key must at a minimum include sensor_id and m_timestamp. As it turns out, {sensor_id, m_timestamp}+ = {sensor_id, m_timestamp, sensor_type, value, marginal_cost}, so {sensor_id, m_timestamp} is also superkey. Due to minimality, it is thus the only key of the relation.

(b) Is `sensor_measurement` in BCNF, 3NF, or neither? Why?

The left sides of the dependencies sensor_id -> sensor_type and sensor_type -> marginal_cost are not composed of superkeys, so the relation is not in BCNF. Additionally, the attributes on the right sides of these dependencies are not prime, so the relation is not in 3NF either.

The engineers of `thriftyiot.com` would like to eliminate as much as possible insertion, update, and deletion anomalies in their database. Now answer the following questions about how triggers could be used for this purpose:

NOTE: For the questions about triggers below, you do not require that you provide the trigger code due to time constraints. However, you should clearly state for each trigger you suggest: 1) whether the trigger is fired in response to which of insert/update/delete events and when; 2) if the trigger has a firing condition or not, providing the corresponding `WHEN` *clause if so; 3) if the trigger is statement-level or for each row; 4) exactly what the trigger code should do when the trigger is fired. Providing the trigger code trivially satisfies the requirements 1)–4), and it is also allowed (just not required).*

(c) Consider the relation `sensor_type` and `sensor`. What triggers should be created over these relations to disallow updates to the `sensor_type` in `sensor` and to the `marginal_cost` in `sensor_type`? That is, once tuples are inserted in these relations, we wish the values provided for the attributes to no longer be changeable.

First trigger: 1) Trigger on sensor before update; 2) condition when :new.sensor_type <> :old.sensor_type; 3) for each row; 4) Raise an exception so as to forbid update.

Second trigger: 1) Trigger on sensor_type before update; 2) condition when :new.marginal_cost <> :old.marginal_cost; 3) for each row; 4) Raise an exception so as to forbid update.

NOTE: One could also create statement-level triggers that forbid all updates, including to the respective primary keys; however, this forbids a bit more than required by the question.

(note that either code or description as above are acceptable solutions)

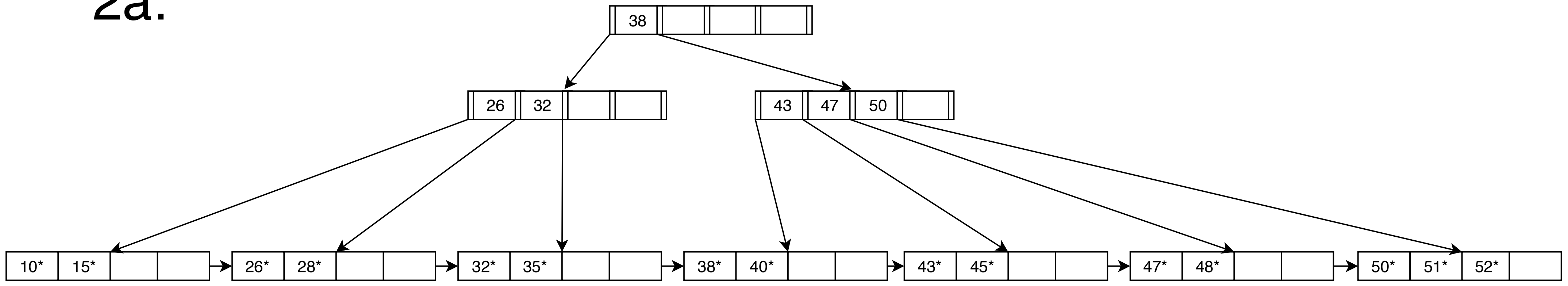
(d) Consider the relation `sensor_measurement`. What trigger (or triggers) should be created to ensure that upon insertion of a new tuple, the `marginal_cost` attribute reflects the corresponding value for the sensor type recorded in `sensor_type` and that the `sensor_type` attribute reflects the corresponding value for the sensor ID recorded in `sensor`?

A single trigger is enough: 1) Trigger on sensor_measurement before insert; 2) no condition clause (WHEN); 3) for each row; 4) Execute the query SELECT sensor_type, marginal_cost FROM sensor_type INNER JOIN sensor USING (sensor_type) WHERE sensor_id = :new.sensor_id, then assign the returned sensor_type and marginal_cost to :new.sensor_type and :new.marginal_cost.

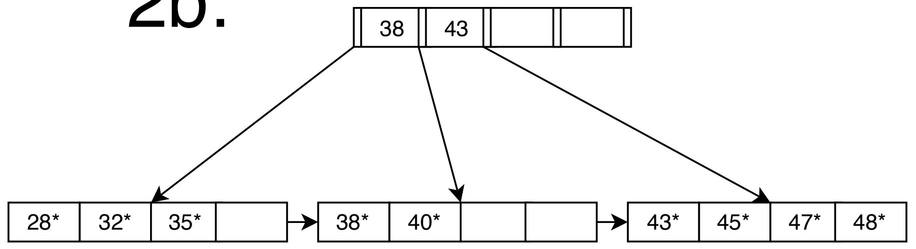
NOTES: If two triggers are created instead of one, there may be indeterminacy on the order of firing, which can create undesired effects. Even with a single trigger, two independent SELECT statements, one on sensor_type and one on sensor can be dangerous if the values provided in :new do not actually join in the other tables (so this needs to be checked in the trigger code).

(note that either code or description as above are acceptable solutions)

Question 2a:



Question 2b:


38
43
28*
32*
35*
38*
40*
43*
45*
47*
48*