Question 1. (25 pts)

Consider the following relations for a database that keeps track of student enrollement in courses. Also consider that the relations are bags that allow duplicates:

- student<id, name, age, deptid> storing 100000 tuples
- department<id, name> storing 600 tuples
- course<id, name, semester> strong 3000 tuples
- enrollment<student_id, course_id, year, grade> storing 4000000 tuples

(a) For the following query:

List id and name of all students who didn't take courses in the first semester of 2021

give two RA expressions for this query (for example, by applying equivalence rules), one of which is optimized in terms of performance, and briefly discuss the estimated performance of the two expressions

```
Select s.id, s.name
From students, enrollment e
Where s.id = e.enrolmentid AND NOT EXISTS
      (SELECT *
       FROM course c
       where c.course_ID = e.courseID
        AND  c.semester = 1 AND e.year = 2021)
```

⇒ Not optimised as there is a big subquery

(A) $\Pi_{s.id, s.name}(\sigma_{c.semester=1 \wedge e.year=2021}(\rho_c(course) \bowtie \rho_e(enrollment) \bowtie \rho_s(student)))$

↳ Not optimised as join THEN filtering, it is better to do the inverse for memory

(B) $\Pi_{s.id, s.name}(\rho_s(student) \bowtie (\sigma_{e.year=2021}(\rho_e(enrollment)) \bowtie \sigma_{c.semester=1}(\rho_c(course))))$

↳ optimised as the filtering is made BEFORE the joins

(A) Filter 100.000 x 3000 x 4000.000 tuples

(B) Filter on 100.000 + 3000 + 4000.000 tuples ⇒ this is non negligible!!

(b) Write an RA expression to express this query:

List all department id and name of the students who attend the Database course ⇒ set-based query

Make sure not to show duplicate results

```
SELECT s.name, s.deptid DISTINCT FROM student s, course c, enrollment e
WHERE c.name = 'Database' AND c.id = e.course_id AND e.student_id = s.id
```

$\Pi_{s.name, s.dept\_id}(\rho_c(\sigma_{name='Database'}(course)) \bowtie_{c.id=e.course\_id} \rho_e(enrollment) \bowtie_{e.student\_id=s.id} \rho_s(student))$

↳ There will be no duplicates as Π is a projection and eradicates them !:P

$\pi_{s.name, s.dept\_id}(\rho_C(\sigma_{name='Database'}(course)) \bowtie_{c.id=e.course\_id} \rho_e(enrollment) \bowtie_{e.student\_id=s.id} \rho_s(student))$

↳ There will be no duplicates as π is a projection and eradicates them :P

(c) Normalize/flatten the following query and give its equivalent RA expression

```
SELECT course.name
FROM course
WHERE course.id in (
      SELECT enrollment.course_id
      FROM student, enrollment
      WHERE student.id = enrollment.student_id AND
      student.name LIKE '%son');
```

Normalizing → transforming to a EXIST subquery

```
SELECT course.name
FROM course
WHERE EXISTS ( SELECT 1 FROM enrollment, student
               WHERE enrollment.student_id = student.id AND student.name like '%son')
```

Flattening is done in 6 steps

1. Translate the subquery

$$\rho_s(\sigma_{name\ like\ '\%son'}(student) \bowtie_{e.student\_id=s.id} \rho_e(enrollment))$$

2. Add context relation and parameters

$$\pi_{e.course\_id}(\rho_s(\sigma_{name\ like\ '\%son'}(student) \bowtie_{e.student\_id=s.id} \rho_e(enrollment)))$$

3. Translate the FROM of the outer query

$$\rho_c(course)$$

4. Synchronize both expression

$$\rho_c(course) \bowtie \pi_{e.course\_id}(\rho_s(\sigma_{name\ like\ '\%son'}(student) \bowtie_{e.student\_id=s.id} \rho_e(enrollment)))$$

5. simplify

6. Complete the expression

$$\pi_{c.name}(\rho_c(course) \bowtie \pi_{e.course\_id}(\rho_s(\sigma_{name\ like\ '\%son'}(student) \bowtie_{e.student\_id=s.id} \rho_e(enrollment))))$$

(d) Suppose relation R<A,B,C, D> has the tuples

(1,2,3,4)
(1,2,3,5)
(3,2,1,0)

(A) Using **bag** projection and theta-join, how many tuples appear in the result of:

$$\pi_{A,B}(R) \bowtie_{R.B<S.B} \rho_S(\pi_{B,C}(R))$$

(B) Using **set** projection and theta-join, how many tuples appear in the result?

A	B	C	D
1	2	3	4
1	2	3	5

(A) $\pi_{A,B}(R)$

R

A	B
1	2
1	2

$\rho_S(\pi_{B,C}(R))$

S

B	C
2	3
2	5

1	2	3	1
1	2	3	5
3	2	1	0

1	2
1	2
3	2

2	3
2	3
2	1

$\Rightarrow \Pi_{A,B}(R) \bowtie_{R.B < S.B} \rho_S(\Pi_{B,C}(R))$ outputs 0 results as . 2 is <2

(B) If it is set based $\Pi_{A,B}(R)$

R

A	B
1	2
3	2

$\rho_S(\Pi_{B,C}(R))$

S

B	C
2	3
2	1

$\Rightarrow$ 0 output

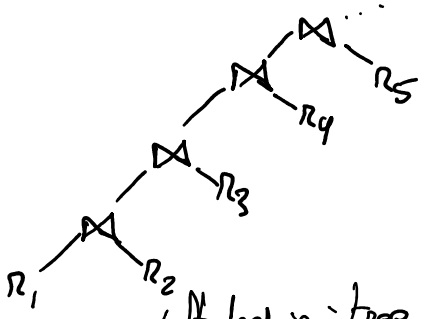
(a) Discuss by giving atleast two reasons why left deep join trees are used, for example in System R, as a heuristic for solving the join ordering optimization problem

The join ordering problem consists in determining the order in which to execute all joins

$$R_1 \bowtie R_2 \bowtie \dots \bowtie R_n$$

with R_i relations that share a common attribute.

A left deep join tree looks like this:



Left deep join tree

$\Rightarrow$ The idea is that it executes a join with always one of the two relation that is already in the memory/computed

$\Rightarrow$ This is used in System R to gain time complexity: it goes from $O(3^n)$ to $O(n2^n)$

$\Rightarrow$ This is also used to reduct the search space!

(b) Which of the following will be stored in the system catalogue of PostgreSQL. Choose zero to all, and give one line discussion on every point explain why you chose or didn't choose it:

1. Materialized views

Yes, they are all stored as it is a tool that helps to faster the query search

2. Attribute statistics for the optimizer

Yes, histograms, cardinality are stored in the database's catalog
$\Rightarrow$ All

3. The optimization rules

Yes, all. To help the optimiser determine the best query plan

4. User tables, such as course, student, enrollment in question 1

Only their names, types, constraints and more but not the values

Only their names, types, constraints and more but not the values

5. The SQL user functions

Yes, their names, input/output data types but NOT the code of it.

6. The B+tree indexes

Yes, names, types, column where applied, etc

7. User defined type

Yes, same : name, attributes i/o functions +

8. The Database schema

Yes, All => it is composed of all the above information.

(c) State wether each of the following points are true about images in System R. Give one line discussion on every point:

1. One may define multiple images on the same relation

TRUE, different states of the DB => ≠ images of a relation thanks to temporality of DB.

2. One may define multiple images on the same attribute

FALSE, image is for a bigger level such as relations as a whole.

3. Image may be declared as unique

True, this means that each row within a specific image of a relation is unique based on certain attributes
=> helps ensure data integrity

4. Defining a clustered image may lead to changing the ordering of the stored tuples in the physical medium

Yes, it changes the order of the stored tuple as system R stores following a certain clustering criterion

5. One may define multiple clustered images on the same relation

False, this would lead to optimisation conflicts

6. One may define multiple unique images on the same relation

Yes, uniqueness is implied within the image and images are for ≠ snapshots of the relation

7. If available, it is always more efficient to access a relation via an image

False, it depends on the nature of the query and the specific requirements

(a) Recall that in a B+tree:

- one tree node is stored in one disk block. In this excercise consider that the block size is 4096 B
- one tree node stores p pointers and p-1 keys

Consider a B+tree index defined over an int16 attribute. Consider also that a tree/disk pointer is 8 bytes. If the relation that contains the indexed attribute has four million tuples, how many disk accessess would be required to answer a query *key= const*, where *const* is a literal value?

16 bits per int $\Rightarrow$ 4096/16 = 256 ints per disk block/node

pointer = 8 bytes

Relation has 4.000.000 tuples $\rightarrow$ how many IO's to query key = const

$p = ? \Rightarrow p \times 8 + (p-1) \times 2 \leq 4096$

$\Leftrightarrow 8p + 2p - 2 \leq 4096$

$\Leftrightarrow p \leq \frac{4098}{10} = 409{,}8 \Rightarrow p = 409 \rightarrow$ fanout

Let's calculate the number of level/order of the B+Tree

Number of ptrs = fanout = p $\Rightarrow$ height = $\lceil \log_p(\#\text{tuple}) \rceil = \lceil \log_{409}(4.000.000) \rceil$

$\Rightarrow X = \log_{409}(4000.000) \Leftrightarrow 409^X = 4.000.00$

$\Rightarrow X \simeq 2$

$\Rightarrow$ 2 levels : 2 disk access

(b) Discuss the truth of each of the following statements.

1. The leaf level of a B+tree index is always a dense index

Yes, it contains all the TID's and the data values to be indexed

$\Rightarrow$ But NO, it can be sparse depending on the deletion/insertion patterns

2. The root level of a B+tree index is always a dense index

False too, " ".

3. Inserting duplicate keys can result in a non-balanced B+tree

Not always but it can happen.

If we add too many duplicate keys, a node could be too full and this can cause balance problem

4. A B+tree index may require a bigger storage size than its relation

True, storing all nodes can sometime take more storage space than the relation itself

$\Rightarrow$ Indexes are not always the best solution!

5. A B+tree index on a string/text attribute cannot be used to answer queries in the form of *key LIKE pattern*

False, they are efficient for this. Maybe less if we need to look at the values in the middle of strings. But for prefix/suffix $\Rightarrow$ super efficient (mostly prefix)

Question 4. (25 pts)

(a) Give a short answer on the following.

1. What is an operator class in PostgreSQL? Give its importance.
2. How is it possible for PostgreSQL to process user types, without knowing them in advance?
3. How is a materialized view different from a table? How does a materialized view affect query performance.
4. How does tuple insertion happen over a distirbuted table using hash-distribution?
5. What is the importance of replicated tables in distributed databases? Why not distribute all tables?

-> Not seen in 2023/2024

1. An operator class in PostgreSQL is used to define/map the operators and support functions to indexes in PostgreSQL
2. NOT SURE but I would say that they are defined as composite (mix of existing types) and defined by the user that specifies storage, i/o functions and statistics.
3. The table stores persistent data, the materialized view stores the result set of a query that has been precomputed

4 (b) not seen this year.