Student ID: 300375123 **First Name: Philip** **Last Name: Chang**

School of Engineering and Computer Science

SWEN 304 **Database System Engineering**

Assignment 2

Due: 23:59, Friday, 9 May 2025

The objective of this assignment is to test your understanding of Relational Algebra and Query Processing and Optimization. It is worth **10%** of your final grade. The assignment is marked out of 100.

In Appendix 1, you will find short recapitulation of formulae needed for cost-based optimization. Appendix 2 contains an abbreviated instruction for using PostgreSQL.

Submission Instructions:

- Please submit your project in **pdf** with your **student ID** and **Name** via the submission system.
- Submissions not in **pdf** will incur **3 marks** deduction from the total marks.

Question 1. Relational Algebra [35 marks]

Consider the Suppliers database schema given below.

Set of relation schemas:

Items ({*Id, Description, Category*}, {*Id*}),
Suppliers ({*SId, Name, Phone, Location*}, {*SId*})
Supplied_By ({*Id, SId, Amount, Year, Price*}, {*Id + SId + Year*})

Set of referential integrity constraints:
Supplied_By [*Id*] $\subseteq$ *Items* [*Id*],
Supplied_By [*SId*] $\subseteq$ *Suppliers* [*SId*]

In this question, you will be given queries on the Suppliers database above in two ways. Firstly, queries are given in plain English and you must answer them in Relational Algebra. Secondly, queries are given in Relational Algebra and you must answer them in plain English and in SQL. Submit all your answers in printed form.

a) [20 marks] Translate the following queries into Relational Algebra:

1) **[5 marks]** Retrieve the names of all suppliers who *always* supply items of category ‘bread’.

πName(Suppliers $\bowtie$ (πSId(Suppliers) − πSId(Supplied_By $\bowtie$ σCategory $\neq$ 'Bread'(Items))))

2) **[5 marks]** Retrieve the descriptions of all items that are supplied by *two or more* suppliers.

πDescription(Items $\bowtie$ πId(σSId $\neq$ S2Id (ρSId$\rightarrow$S2Id, Id$\rightarrow$S2IdItem(Supplied_By) $\bowtie$ Supplied_By $\bowtie$ S2IdItem = Id)))

3) **[5 marks]** Retrieve the names of all suppliers who have *not* supplied any item in 2025.

πName(Suppliers $\bowtie$ (πSId(Suppliers) − πSId(σYear = 2025(Supplied_By))))

4) **[5 marks]** Retrieve the descriptions of all items that have been supplied by suppliers in Nelson who *never* supply items with a price higher than $200.

πDescription(Items $\bowtie$ πId(Supplied_By $\bowtie$ (πSId(Suppliers $\bowtie$ σLocation = 'Nelson'(Suppliers)) − πSId(σPrice > 200(Supplied_By)))))

b) [15 marks] Translate the following queries into plain English and into SQL:

1) [5 marks] $\pi_{Name,\ Phone,\ Category}$ (*Items* * ($\sigma_{Year=2025}$ (*Supplied_By*) * *Suppliers*))

Retrieve the name, phone number, and category of items for each supplier who supplied items in the year 2025.

```
SELECT DISTINCT s.Name, s.Phone, i.Category
FROM Items i
JOIN Supplied_By sb ON i.Id = sb.Id
JOIN Suppliers s ON sb.SId = s.SId
WHERE sb.Year = 2025;
```

2) [5 marks] $\pi_{Description,\ Name}$ ($\sigma_{price<10}$ (*Items* * (*Supplied_By* * *Suppliers*)))

Retrieve the description of items and name of suppliers who supplied those items at a price lower than $10.

```
SELECT DISTINCT i.Description, s.Name
FROM Items i
JOIN Supplied_By sb ON i.Id = sb.Id
JOIN Suppliers s ON sb.SId = s.SId
WHERE sb.Price < 10;
```

3) [5 marks] π_{SId} (*Supplied_By* * ($\sigma_{Category='Meat'}$ (*Items*))) $\cap$ π_{SId} ($\sigma_{Amount>1000}$ (*Supplied_By*))

Retrieve the IDs of all suppliers who have both supplied meat items and have supplied an item in a quantity greater than 1000.

```
SELECT DISTINCT sb1.SId
FROM Supplied_By sb1
JOIN Items i ON sb1.Id = i.Id
WHERE i.Category = 'Meat'
WHERE sb.Price < 10;
INTERSECT
SELECT DISTINCT sb2.Sid
FROM Supplied_By sb2
WHERE sb2.Amount > 1000;
```

Question 2. Heuristic and Cost-Based Query Optimization [43 marks]

The DDL description of a part of the University database schema is given below.

```
CREATE DOMAIN StudIdDomain AS int NOT NULL CHECK (VALUE >= 30000000 AND
VALUE <= 300099999);

CREATE DOMAIN CharDomain AS char(15) NOT NULL;

CREATE DOMAIN NumDomain AS smallint NOT NULL CHECK (VALUE BETWEEN 0 AND
10000);

CREATE TABLE Student (
StudentId StudIdDomain PRIMARY KEY,
Name CharDomain,
NoOfPts NumDomain CHECK (NoOfPts < 1000),
Tutor StudIdDomain REFERENCES Student(StudentId)
);

CREATE TABLE Course (
CourseId CharDomain PRIMARY KEY,
CourName CharDomain,
ClassRep StudIdDomain REFERENCES Student(StudentId)
);

CREATE TABLE Enrolled (
StudentId StudentIdDomain REFERENCES Student,
CourseId CharDomain REFERENCES Course,
Term NumDomain CHECK(Term BETWEEN 2000 AND 2100),
Grade CharDomain CHECK (Grade IN ('A+', 'A', 'A-', 'B+', 'B', 'B-', 'C+',
'C')),
PRIMARY KEY (StudentId, CourseId, Term)
);
```

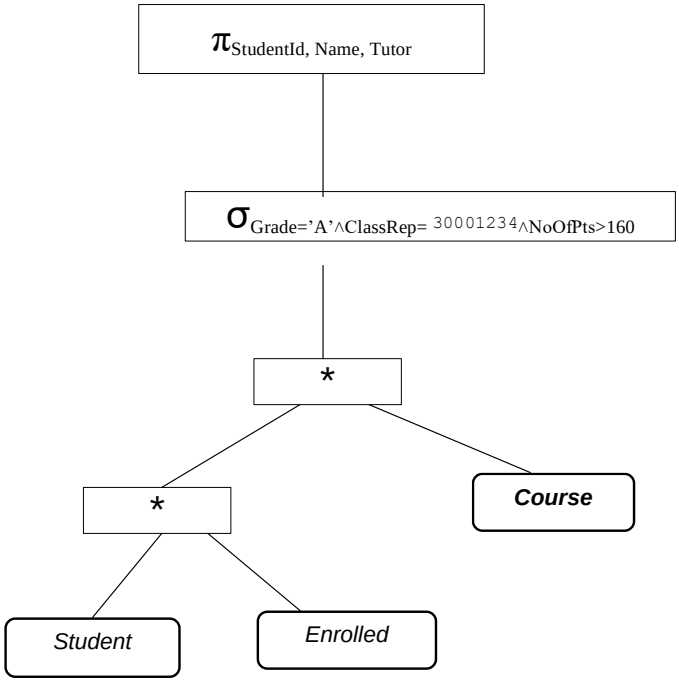
a) [18 marks] Heuristic query optimization

1) [4 marks] Transfer the following query into Relational Algebra.

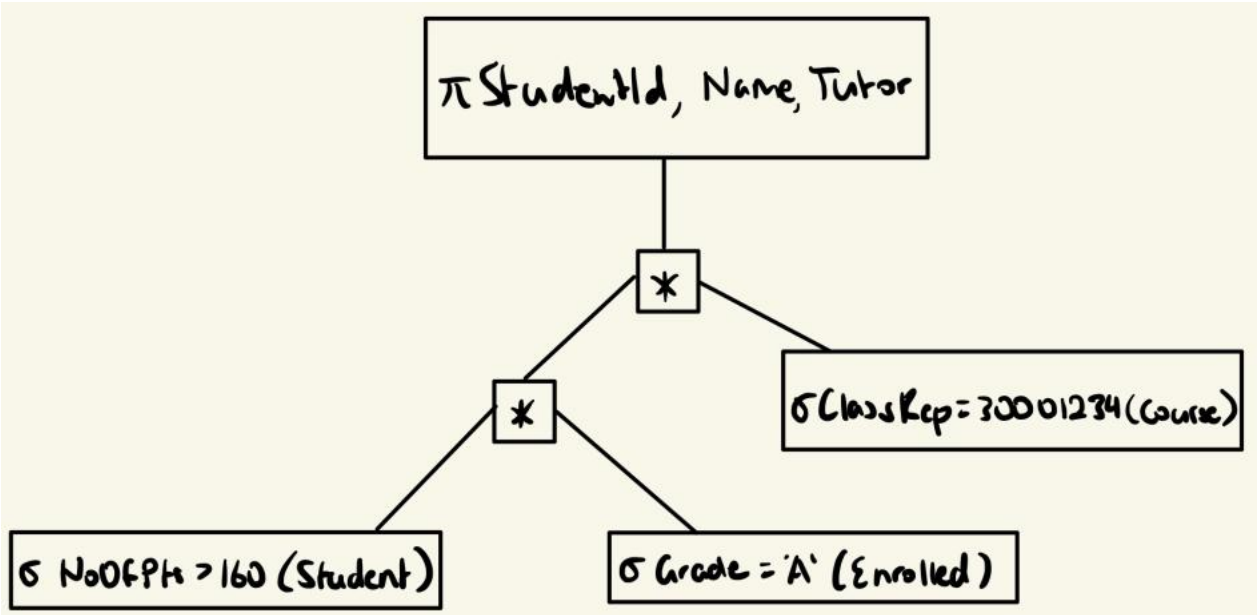
```
SELECT StudentId, Name, Grade

FROM Student NATURAL JOIN Enrolled NATURAL JOIN Course

WHERE CourName = 'Database Systems' AND Term = 2025
                AND Tutor = 'Tom';
```

πStudentId, Name, Grade (σCourName = 'Database Systems' ∧ Term = 2025 ∧ Tutor = 'Tom' (Student ⋈ Enrolled ⋈ Course))

2)[14 marks] Transfer the following query tree into an optimized query tree using the query optimization heuristics. Draw the optimised query tree, and list the heuristic rules you applied.



ANSWER



To optimize this query tree, I used the following heuristic rules:
- Push down selections.
- Push down projections.
- Reorder joins.
- Replace cartesian products with joins.

b) [25 marks] Query cost calculation

Suppose the following:

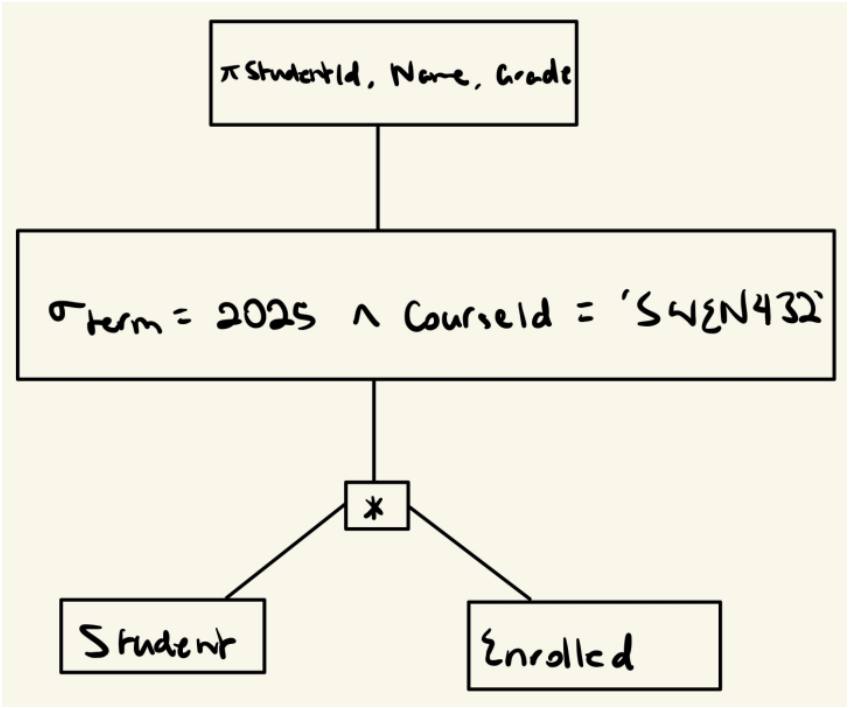
- The *Student* relation contains data about $n_s = 50000$ students (enrolled during the past *10* years),
- The *Course* relation contains data about $n_c = 1000$ courses,
- *The Enrolled relation contains data about $n_e = 400{,}000$ enrollments,*
- *All data distributions are uniform (i.e. each year approximately the same number of students enrolls into each course),*
- *The intermediate results of the query evaluation are materialized,*
- *The final result of the query is materialized.*

Note: If you feel that some information is missing, please make a reasonable assumption and make you assumption explicit in your answer.

For each of the given two queries below ***draw a query tree*** and ***calculate the cost of executing query.***

(i) $\pi_{\text{StudentId, Name, Grade}}(\sigma_{\text{term = 2025} \wedge \text{CourseId = 'SWEN432'}}(\text{Student} \star \text{Enrolled}))$

ANSWER



In my query tree, I have a join between Student and Enrolled, followed by a selection on term = 2025 and CourseId = 'SWEN432'.

Here I will assume that a tuple size is around 120 bytes per tuple. With 400,000 tuples, we can calculate the join cost to be about ~48MB.

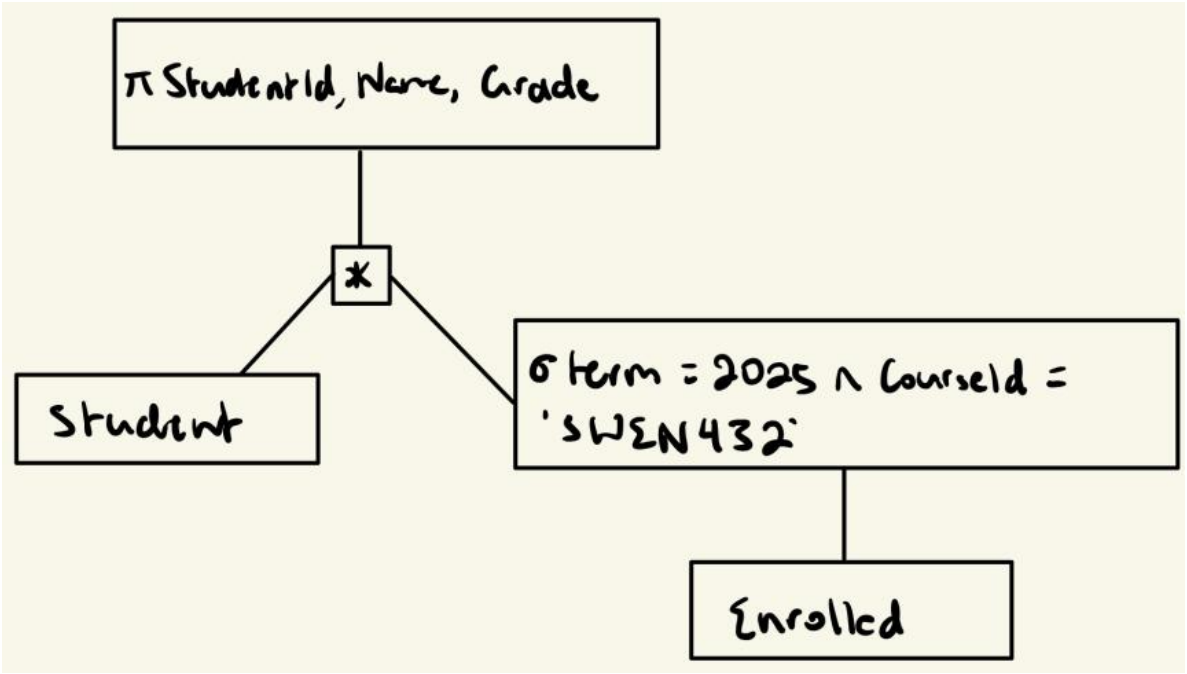
Then, the selection cost consists of 10 years of data meaning 2025 is 10%. We have 1000 courses, SWEN432 is 1/1000. So combined selectivity = 0.1 * 0.001 = 0.0001 (0.01%).

Then we have filtered size, 400,000 * 0.0001 = 40 tuples. Each tuple approx. 120 bytes which is relatively cheap due to the filtering.

As we are only keeping StudentId, Name, and Grade, let's say we are left with ~60 bytes per tuple. The final resulting size is 40 tuples * 60 bytes = 2,400 bytes which is around 2.4KB.

(ii) $\pi_{\text{StudentId, Name, Grade}}$ (Student $*$ $\sigma_{\text{term = 2025} \wedge \text{CourseId = 'SWEN432'}}$ (Enrolled))

ANSWER



This query tree first applies a selection on the Enrolled relation to filter rows for term = 2025 and CourseId = 'SWEN432'. The filtered rows are then joined with the Student relation. Finally, a projection extracts only the StudentId, Name, and Grade attributes.

Using the same assumptions as above, I will apply the same combined selectivity of 0.0001, and a filtered result of 400,000 * 0.0001 = 40 tuples. The selection is applied before the join, so only 40 rows from Enrolled are joined.

If I use the same 1 tuple = 120 bytes, then the total size of the materialized join is 40 * 120 = 4,800 bytes, which is approx.. 4.8KB.

Following the join, the projection step extracts only our three required attributes (StudentId, Name, Grade). Assuming a projected tuple size of approx. 60 bytes again, the size of the final result becomes 40 * 60 bytes = 2,400 bytes, which is roughly 2.4KB.

iii) Which of the above two trees has a smaller query cost and why?

ANSWER

This query
2b(ii) likely has a smaller query cost as the first query performs student ⋈ enrolled, materializing 400,000 rows then applies a selection, while the second query applies the selection on Enrolled first, reducing it before later joining with Student. This avoids creating the large intermediate 400,000-row join result.

The second query in 2b(ii) would do less work (smaller intermediate result), as it stores fewer tuples during execution and has overall lower database engine processes.

Hint: *To find out about the sizes of attributes in PostgreSQL please consult the documentation (www.postgresql.org/docs/9.2/static/datatype.html) or check this tutorial (www.tutorialspoint.com/postgresql/postgresql_data_types.htm).*

Note: *Use the formulae introduced in the lecture notes (also in Appendix) to compute the estimated query costs. Total query cost of a query tree is the sum of the costs of all leaves, the intermediate notes and the root of a query tree.*

Question 3. PostgreSQL and Query Optimization [22 marks]

You are asked here to improve efficiency of two database queries. The only condition is that after making improvements your queries produce the same results as the original ones, and your databases contain the same information as before.

For the optimization purposes, you will use two databases. A database that was dumped into the file

`GiantCustomer.data`

And the other database that was dumped into the file

`Library.data`

Both files are accessible from the course Assignments web page. Copy both files into your private directory. You are to:

i. Use PostgreSQL in order to create a database and to execute the command

```
psql –d <database_name> -f ~/<file_name>
```

This command will execute the `CREATE TABLE` and `INSERT` commands stored in the file `<file_name>`, and make a database for you.

ii. Execute the following commands:

- `VACUUM ANALYZE customer;`

on the database containing `GiantCustomer.data` file, and

- `VACUUM ANALYZE customer;`
- `VACUUM ANALYZE loaned_book;`

on database containing `Library.data` file.
These commands will initialize the catalog statistics of your database `<database_name_x>`, and allow the query optimizer to calculate costs of query execution plans.

iii. Read the PostgreSQL Manual and learn about `EXPLAIN` command, since you will need it when optimizing queries. Note that a PostgreSQL answer to `EXPLAIN <query>` command looks like:

```
NOTICE:  QUERY PLAN:

Merge Join  (cost=6.79..7.10 rows=1 width=24)
  ->  Sort  (cost=1.75..1.75 rows=23 width=12)
        ->  Seq Scan on cust_order o  (cost=0.00..1.23 rows=23 width=12)
  ->  Sort  (cost=5.04..5.04 rows=2 width=12)
        ->  Seq Scan on order_detail d  (cost=0.00..5.03 rows=2 width=12)
```

Here, PosgreSQL is informing you that it decided to apply Sort Merge Join algorithm and that this join algorithm requires Sequential Scan and Sort of both relations. The shaded number `7.10` is an estimate of the query execution cost made by PostgreSQL. When making an

improved query, you will compare your achievement to this figure, and compute the relative improvement using the following formula

(original_cost – new_cost) / original_cost.

You may also want to use `EXPLAIN ANALYZE <query>` command that will give you additional information about the actual query execution time. Please note, the query execution time figures are not quiet reliable. They can vary from one execution to the other, since they strongly depend on the workload imposed on the database server by users. ***To get a more reliable query time measurement, you should run your query a number of times and then calculate the average***.

a) [6 marks] Improve the cost estimate of the following query:

```
select count(*) from customer where no_borrowed = 4;
```

issued against the database containing `GiantCustomer.data`. Make such changes to your database or to the query that will allow you to produce the same result as the original query, but in a more efficient way. Of course, your changes have to be fair. Analyze the output from the PostgreSQL query optimizer and make a plan on how to improve the efficiency of the query. *Show what you have done by copying appropriate messages from the PostgreSQL prompt and explain why you have done it, calculate the improvement*. Each time you want to quit with that database, please drop it, since it occupies a lot of memory space.

Marking schedule:
You will receive:

- 5 marks if your query cost estimate is at least 64% better than the original one.
- between 2 and 4 marks if your query cost estimate is between 20% and 60% better than the original one and your marks will be calculated proportionally.
- up to 1 additional marks if you give reasonable explanations of what you have done.

ANSWER

Before Optimization:

```
changphila02=> EXPLAIN ANALYZE SELECT count(*) FROM customer WHERE no_borrowed = 4;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=115.36..115.37 rows=1 width=8) (actual time=2.042..2.046 rows=1 loops=1)
   ->  Seq Scan on customer  (cost=0.00..114.25 rows=443 width=0) (actual time=0.024..1.599 rows=443 loops=1)
         Filter: (no_borrowed = 4)
         Rows Removed by Filter: 4537
 Planning Time: 0.718 ms
 Execution Time: 2.271 ms
(6 rows)
```

We can see that running the original query with EXPLAIN ANALYZE performed a sequential scan on all 4980 rows, with 443 matches. This was inefficient with a cost of 114.25.

To improve optimization, I used index-based access on the no_borrowed attribute to avoid scanning the full table. I created an index to do this.

After Optimization:

```
changphila02=> CREATE INDEX idx_no_borrowed ON customer(no_borrowed);
CREATE INDEX
changphila02=> VACUUM ANALYZE customer;
VACUUM
changphila02=> EXPLAIN ANALYZE SELECT count(*) FROM customer WHERE no_borrowed = 4;
                                                            QUERY PLAN

-------------------------------------------------------------------------------------------------------------
------------------
 Aggregate  (cost=13.14..13.15 rows=1 width=8) (actual time=3.254..3.258 rows=1 loops=1)
   ->  Index Only Scan using idx_no_borrowed on customer  (cost=0.28..12.04 rows=443 width=0) (actual time=2.319..2.806
rows=443 loops=1)
         Index Cond: (no_borrowed = 4)
         Heap Fetches: 0
 Planning Time: 0.151 ms
 Execution Time: 3.349 ms
(6 rows)
```

Here we can see that after performing an index-only scan, the logical I/O and CPU usage were both drastically reduced. We can see that it did not need to fetch heap pages (Heap Fetches: 0)

Pre-index: 114.25
Post-index: 12.04

(114.25 – 12.04) / 114.25 ≈ 0.895
89.5% improvement.

If we compare the total cost before and after, we can see that it results in an 89.5% improvement after creating an index on the no_borrowed column, reducing query cost.

b) [4 marks] Improve the efficiency of the following query:

```
select * from customer where customerid = 2446;
```

issued against the database containing `GiantCustomer.data`. Make such changes to your database or to the query that will allow you to produce the same result as the original query, but in a more efficient way. Analyze the output from the PostgreSQL query optimizer and make a plan how to improve the efficiency of the query.
Show what you have done by copying appropriate messages from the PostgreSQL prompt and explain why you have done it, calculate the improvement. Each time you want to quit with that database, please drop it, since it occupies a lot of memory space.

Marking schedule:
You will receive

- 3 marks if your query cost estimate is 93% (or more) better than the original one.
- between 1 and 3 marks if your query cost estimate is better between 20% and 93% than the original one and your marks will be calculated proportionally to the improvement achieved.
- up to 1 additional marks if you give reasonable explanations of what you have done.

Before Optimization:

```
changphila02=> EXPLAIN ANALYZE SELECT * FROM customer WHERE customerid = 2446;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on customer  (cost=0.00..114.25 rows=1 width=56) (actual time=0.435..0.968 rows=1 loops=1)
   Filter: (customerid = 2446)
   Rows Removed by Filter: 4979
 Planning Time: 0.936 ms
 Execution Time: 1.110 ms
(5 rows)
```

We can see that prior to optimization; this query ran a sequential scan over all 4980 rows in the customer table which is inefficient for a targeted lookup on a specific customer ID.

Creating an Index:

```
changphila02=> \d customer
                Table "public.customer"
   Column    |     Type      | Collation | Nullable | Default
-------------+---------------+-----------+----------+---------
 customerid  | integer       |           | not null |
 l_name      | character(15) |           | not null |
 f_name      | character(15) |           |          |
 city        | character(15) |           |          |
 no_borrowed | integer       |           |          | 0
Indexes:
    "idx_no_borrowed" btree (no_borrowed)
Check constraints:
    "customer_city" CHECK (city = 'Wellington'::bpchar OR city = 'Upper Hutt'::bpchar OR city = 'Lower Hutt'::bpchar)
    "customer_customerid" CHECK (customerid > 0)
```

To reduce the cost of scanning, I added an index on the customer column, allowing PostgreSQL to directly locate the relevant row using an index structure.

After Optimization:

```
changphila02=> CREATE INDEX idx_customerid ON customer(customerid);
CREATE INDEX
changphila02=> VACUUM ANALYZE customer;
VACUUM
changphila02=> EXPLAIN ANALYZE SELECT * FROM customer WHERE customerid = 2446;
                                                      QUERY PLAN

------------------------------------------------------------------------------------------------------------------------
--
 Index Scan using idx_customerid on customer  (cost=0.28..8.30 rows=1 width=56) (actual time=0.150..0.153 rows=1 loops=1
)
   Index Cond: (customerid = 2446)
 Planning Time: 0.162 ms
 Execution Time: 0.173 ms
(4 rows)
```

After

Pre-index: 114.25
Post-Index: 8.30

(114.25 – 8.30) / 114.25 ≈ 0.927
92.7% improvement.

If we compare the before and after costs, we can see that there was a resulting improvement of 92.7%. This is due to using an index scan rather than a sequential scan.

c) [12 marks] The following query is issued against the database containing the data from `Library.data`. It retrieves information about every customer for whom there exist less than three other customers borrowing more books than she/he did:

```
select clb.f_name, clb.l_name, noofbooks
from (select f_name, l_name, count(*) as noofbooks
     from customer natural join loaned_book
     group by f_name, l_name) as clb
     where 3 > (select count(*)
                from (select f_name, l_name, count(*) as noofbooks
                      from customer natural join loaned_book
                      group by f_name, l_name) as clb1
                      where clb.noofbooks<clb1.noofbooks)
                      order by noofbooks desc;
```

Unfortunately, the efficiency of the given query is very poor. Make such changes to your database or to the query that will allow you to produce the same result as the original query, but in a more efficient way.
Show what you have done by copying appropriate messages from the PostgreSQL prompt, calculate the improvement, and briefly explain why the query given is inefficient and why your query is better.

Marking schedule:

You will receive:

- 3 marks if you explain in English how the query computes the answer,
- 5 marks if your query has a cost estimate 70% (or more) better than the original one (otherwise, your marks will be calculated proportionally to the improvement achieved),
- An additional 2 marks if you give reasonable explanations of why the query given is inefficient and why is your query better.

ANSWER

The initial given query is inefficient due to the subquery clb1 duplicating inside the WHERE clause to calculate borrow counts. This means that PostgreSQL executes the same group by subquery once for the outer table (clb), then once again for every row inside the subquery (clb1), resulting in $O(n^2)$ behavior if there are n customers.

Original Query:

```
changphilLib=> EXPLAIN ANALYZE
changphilLib-> SELECT clb.f_name, clb.l_name, noofbooks
changphilLib-> FROM (
changphilLib(>   SELECT f_name, l_name, COUNT(*) AS noofbooks
changphilLib(>   FROM customer NATURAL JOIN loaned_book
changphilLib(>   GROUP BY f_name, l_name
changphilLib(> ) AS clb
changphilLib-> WHERE 3 > (
changphilLib(>   SELECT COUNT(*)
changphilLib(>   FROM (
changphilLib(>     SELECT f_name, l_name, COUNT(*) AS noofbooks
changphilLib(>     FROM customer NATURAL JOIN loaned_book
changphilLib(>     GROUP BY f_name, l_name
changphilLib(>   ) AS clb1
changphilLib(>   WHERE clb.noofbooks < clb1.noofbooks
changphilLib(> )
changphilLib-> ORDER BY noofbooks DESC;
                                                                   QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
------------------------------
 Sort  (cost=83.02..83.04 rows=8 width=40) (actual time=0.973..1.001 rows=3 loops=1)
   Sort Key: clb.noofbooks DESC
   Sort Method: quicksort  Memory: 25kB
   ->  Subquery Scan on clb  (cost=3.05..82.90 rows=8 width=40) (actual time=0.742..0.926 rows=3 loops=1)
         Filter: (3 > (SubPlan 1))
         Rows Removed by Filter: 12
         ->  HashAggregate  (cost=3.05..3.28 rows=23 width=40) (actual time=0.339..0.367 rows=15 loops=1)
               Group Key: customer.f_name, customer.l_name
               Batches: 1  Memory Usage: 24kB
               ->  Hash Join  (cost=1.52..2.86 rows=26 width=32) (actual time=0.195..0.285 rows=26 loops=1)
                     Hash Cond: (loaned_book.customerid = customer.customerid)
                     ->  Seq Scan on loaned_book  (cost=0.00..1.26 rows=26 width=4) (actual time=0.018..0.045 rows=26 lo
ops=1)
                     ->  Hash  (cost=1.23..1.23 rows=23 width=36) (actual time=0.126..0.129 rows=23 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 10kB
                           ->  Seq Scan on customer  (cost=0.00..1.23 rows=23 width=36) (actual time=0.041..0.068 rows=2
3 loops=1)
         SubPlan 1
           ->  Aggregate  (cost=3.44..3.45 rows=1 width=8) (actual time=0.031..0.033 rows=1 loops=15)
                 ->  HashAggregate  (cost=3.05..3.34 rows=8 width=40) (actual time=0.019..0.025 rows=4 loops=15)
                       Group Key: customer_1.f_name, customer_1.l_name
                       Filter: (clb.noofbooks < count(*))
                       Batches: 1  Memory Usage: 24kB
                       Rows Removed by Filter: 11
                       ->  Hash Join  (cost=1.52..2.86 rows=26 width=32) (actual time=0.128..0.217 rows=26 loops=1)
                             Hash Cond: (loaned_book_1.customerid = customer_1.customerid)
                             ->  Seq Scan on loaned_book loaned_book_1  (cost=0.00..1.26 rows=26 width=4) (actual time=0
.004..0.029 rows=26 loops=1)
                             ->  Hash  (cost=1.23..1.23 rows=23 width=36) (actual time=0.073..0.076 rows=23 loops=1)
                                   Buckets: 1024  Batches: 1  Memory Usage: 10kB
                                   ->  Seq Scan on customer customer_1  (cost=0.00..1.23 rows=23 width=36) (actual time=
0.005..0.032 rows=23 loops=1)
 Planning Time: 1.995 ms
 Execution Time: 1.614 ms
(30 rows)
```

The original version of the query performs a nested aggregation in a subquery for each row, leading to duplicated work and quadratic complexity.

Revised Query:

```
changphilLib=> EXPLAIN ANALYZE
changphilLib-> SELECT f_name, l_name, noofbooks
changphilLib-> FROM (
changphilLib(>   SELECT f_name, l_name, COUNT(*) AS noofbooks,
changphilLib(>          RANK() OVER (ORDER BY COUNT(*) DESC) AS rank
changphilLib(>   FROM customer NATURAL JOIN loaned_book
changphilLib(>   GROUP BY f_name, l_name
changphilLib(> ) AS ranked
changphilLib-> WHERE rank <= 3
changphilLib-> ORDER BY noofbooks DESC;
                                                                 QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------------
----------------
 Subquery Scan on ranked  (cost=3.80..4.49 rows=8 width=40) (actual time=0.424..0.518 rows=3 loops=1)
   Filter: (ranked.rank <= 3)
   Rows Removed by Filter: 12
   ->  WindowAgg  (cost=3.80..4.20 rows=23 width=48) (actual time=0.421..0.494 rows=15 loops=1)
         ->  Sort  (cost=3.80..3.86 rows=23 width=40) (actual time=0.402..0.433 rows=15 loops=1)
               Sort Key: (count(*)) DESC
               Sort Method: quicksort  Memory: 26kB
               ->  HashAggregate  (cost=3.05..3.28 rows=23 width=40) (actual time=0.309..0.342 rows=15 loops=1)
                     Group Key: customer.f_name, customer.l_name
                     Batches: 1  Memory Usage: 24kB
                     ->  Hash Join  (cost=1.52..2.86 rows=26 width=32) (actual time=0.164..0.261 rows=26 loops=1)
                           Hash Cond: (loaned_book.customerid = customer.customerid)
                           ->  Seq Scan on loaned_book  (cost=0.00..1.26 rows=26 width=4) (actual time=0.018..0.045 rows
=26 loops=1)
                           ->  Hash  (cost=1.23..1.23 rows=23 width=36) (actual time=0.089..0.097 rows=23 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 10kB
                                 ->  Seq Scan on customer  (cost=0.00..1.23 rows=23 width=36) (actual time=0.021..0.049
rows=23 loops=1)
 Planning Time: 1.486 ms
 Execution Time: 0.949 ms
(18 rows)
```

This new query uses a window function, allowing PostgreSQL to compute all ranks in a single pass and efficiently filter the top 3, reducing complexity to O(n log n) due to the sort.

Original Query: 83.04
Optimized Query: 4.49

(83.04 – 4.49) / 83.04 ≈ 0.946
94.6% improvement.

It is more efficient as the original query runs the same aggregation twice, once in the main query and once per row in the subquery. However, the optimized query I revised replaces this with a single aggregation and ranking, reducing computation and allowing for better planning and indexing.

++++++++++++++++++++++++++++++++

Appendix 1: Formulae for Computing a Query Cost Estimate

For a relation with schema $R = \{A_1,...,A_k\}$, the average size of a tuple is: $r = \sum_{j=1}^{k} l_j$

The size of relation is $s = n \cdot r$, with n as the average number of tuples in the relation,

Select: for a selection node σ_C the assigned size is $a_C \cdot s$, where s is the size assigned to the successor and $100 \cdot a_C$ is the average percentage of tuples satisfying C

Project: for a projection node π_{Ri} the assigned size is $(1 - C_i) \cdot s \cdot r_i / r$, where r_i (r) is the average size of a tuple in a relation over R_i (R), s is the size assigned to the successor and C_i is the probability that two tuples coincide on R_i

Join: for a join node the assigned size is $s_1/r_1 \cdot p \cdot s_2/r_2 \cdot (r_1 + r_2 - r)$, where s_i are the sizes of the successors, r_i are the corresponding tuple sizes, r is the size of a tuple over the common attributes and p is the matching probability

Union: for a union node the assigned size is $s_1 + s_2 - p \cdot s_1$ with the probability p for tuple of R_1 to coincide with a tuple over R_2

Difference: for a difference node the assigned size is $s_1 \cdot (1 - p)$, where $(1 - p)$ is probability that tuple from R_1-relation does not occur as tuple in R_2-relation

Appendix 2: Using PostgreSQL on the workstations

We have a command line interface to PostgreSQL server from ECS, so you need to run it from a terminal.

To connect to the servers of ECS, such as **greta-pt.ecs.vuw.ac.nz** or **barretts.ecs.vuw.ac.nz**, remotely, you can access PostgreSQL server at home via SSH as below:

> **ssh** [username]**@greta-pt.ecs.vuw.ac.nz**

- If you are not asked to enter your password, type "kinit [username]" at the shell prompt and enter your password.

To enable the various applications required, type either

> need swen304tools

or

> need postgresql

You may wish to add either "`need swen304tools`", or the "`need postgresql`" command to your `.cshrc` file so that it is run automatically. Add this command after the command `need SYSfirst`, which has to be the first `need` command in your `.cshrc` file.

There are several commands you can type at the unix prompt:

> createdb ⟨database_name⟩

Creates an empty database. The database is stored in the same PostgreSQL server used by all the students in the class. Your database may have an arbitrary name, but we recommend to name it either `userid` or `userid_x`, where `userid` is your ECS user name and `x` is a number from 0 to 9. To ensure security, you must issue the following command as soon as you log-in into your database for the first time:

```
REVOKE CONNECT ON DATABASE <database_name> FROM PUBLIC;
```

You only need to do this once (unless you get rid of your database to start again). **Note**, your markers may check whether you have issued this command and if they find you didn't, you may be **penalized**.

> **psql** [**–d** ⟨db name⟩]

Starts an interactive SQL session with PostgreSQL to create, update, and query tables in the database. The db name is optional (unless you have multiple databases)

> dropdb ⟨databas_name⟩

Gets rid of a database. (In order to start again, you will need to create a database again)

> pg_dump -i ⟨databas_name⟩ > ⟨file_name⟩

Dumps your database into a file in a form consisting of a set of SQL commands that would reconstruct the database if you loaded that file.

> **psql -d** <database_name> **-f** <file_name>

Copies the file <file_name> into your database <database_name>.

Inside and interactive SQL session, you can type SQL commands. You can type the command on multiple lines (note how the prompt changes on a continuation line). End commands with a ';'

There are also many single line PostgreSQL commands starting with '\' . No ';' is required. The most useful are

\? to list the commands,

\i 〈file_name〉

loads the commands from a file (e.g., a file of your table definitions or the file of data we provide).

\dt to list your tables.

\d 〈table_name〉 to describe a table.

\q to quit the interpreter

copy <table_name> **to** <file_name>
Copy your table_name data into the file file_name.

copy <table_name> **from** <file_name>
Copy data from the file file_name into your table table_name.

Note also that the PostgreSQL interpreter has some line editing facilities, including up and down arrow to repeat previous commands.
For longer commands, it is safer (and faster) to type your commands in an editor, then paste them into the interpreter!