Student ID:300442776 **First Name: Yun** **Last Name: Zhou**

School of Engineering and Computer Science

SWEN 439 **Database System Engineering**

Assignment 2

Due: 23:59, Friday, 6 May 2022

The objective of this assignment is to test your understanding of Relational Algebra and Query Processing and Optimization. It is worth **5%** 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 of SWEN439:
- Submissions not in pdf will incur **3 marks** deduction from the total marks.

Question 1. Relational Algebra [40 marks]

Consider the Suppliers database schema given below.

Set of relation schemas:

Products ({*PId, Description, Category*}, {*PId*}),
Company ({*CId, Name, Phone, Location*}, {*CId*})
Supplied_By ({*PId, CId, Amount, Year, Price*}, {*PId + CId + Year*})

Set of referential integrity constraints:
Supplied_By [*PId*] $\subseteq$ *Products* [*PId*],
Supplied_By [*CId*] $\subseteq$ *Company* [*CId*]

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) [28 marks] Translate the following query into Relational Algebra:

1) [5 marks] For all products of category 'meat' list their descriptions and the names of their supplying companies.

πDescription,Name(σcategory= 'meat'(Products*(Supplied_By*Company)))

2) [5 marks] Retrieve the names of all companies who *always* supply products of category 'fruit'.

πName(σcategory= 'fruit'(Product*(Supplied_By*Company)))

3) [5 marks] Retrieve the descriptions of all products that are supplied by *two or more* companies.

πDescription(πPid(σCid <> OtherCid (σCid -> OtherCid(πPid,Cid(Supplied_By*(πPid,Cid(Supplied_By)*Products))))))

4) [5 marks] Retrieve the names of companies who have *not* supplied any product in 2022.

πName(Company) - πName(σYear='2022'(Products*(Supplied_By*Company)))

5) [8 marks] Retrieve the names of companies who have supplied *all* the products.

CIdπName(σprodCount = count(PId)(Product) (Products*(Supplied_By*Company)* (count(PId)(Product) -> prodCount))))

b) [12 marks] Translate the following queries into plain English and into SQL:

1) $\pi_{Name, Phone}$ (*Products* * ($\sigma_{Amount>1000}$ (*Supplied_By*) * *Company*))

Retrieve names and phones of companies that have more than 1000 supplied amount.

```
SELECT Name,Phone
FROM Supplied_By NATURAL JOIN Products NATURAL JOIN Company
WHERE Amount>1000;
```

2) π_{CId} ($\sigma_{Amount>1000}$ (*Supplied_By*)) ∩ π_{CId} (*Supplied_By* * ($\sigma_{Description='Cake'}$ (*Products*)))

Retrieve Cids of all companies where supplied more than 100 amount of cakes.

```
SELECT CId
FROM Supplied_By NATURAL JOIN Products NATURAL JOIN Company
WHERE Amount > 1000 AND Description= 'Cake';
```

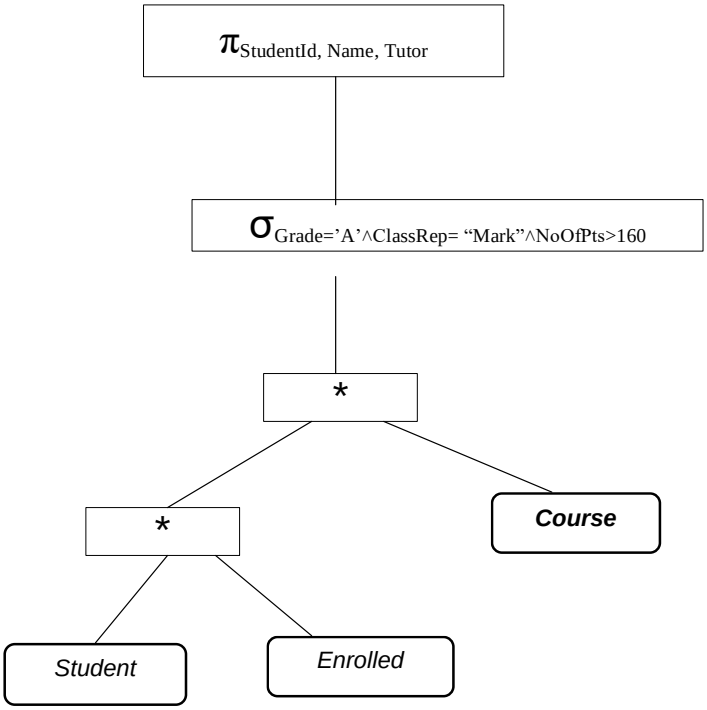
Question 2. Heuristic and Cost-Based Query Optimization [40 marks]

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

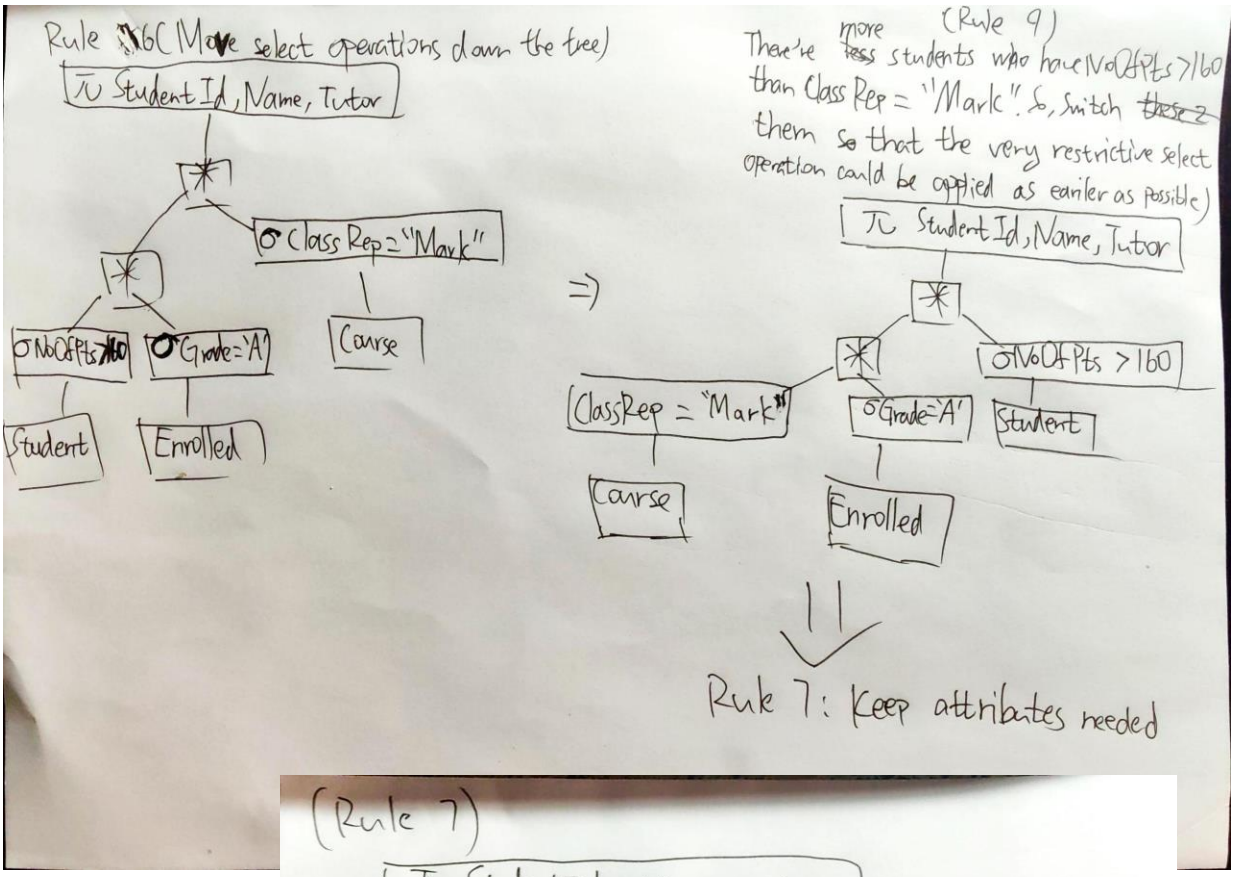
```
CREATE DOMAIN StudIdDomain AS int NOT NULL CHECK (VALUE >= 30000000 AND
VALUE <= 300099999);

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)
);
```

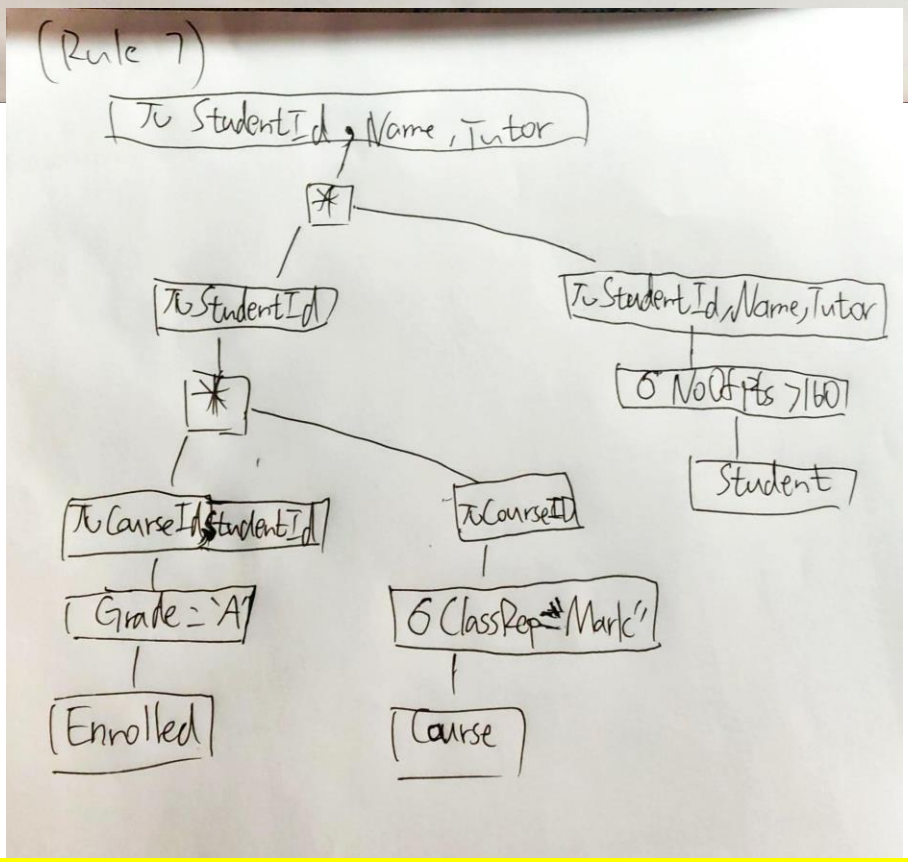
a) [20 marks] Heuristic query optimization

Transfer the following query tree into an optimized query tree using the query optimization heuristics.



ANSWER





Due to the type of ClassRep should be StudIdDomain, which should be the int value, but for this one, the type of “Mark” is not int, which will report error.

b) [20 marks] Query cost calculation

Suppose the following:

- The *Student* relation contains data about $n_s = 50{,}000$ 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) $\pi_{\text{StudentId, Name, Grade}}\ (\sigma_{\text{term = 2022} \wedge \text{CourseId = 'SWEN304'}}\ (\text{Student} \star \text{Enrolled}))$

ANSWER

Through the calculation, the total cost is: 38,453,640 bytes. The assumption and the calculation screenshots are shown below

Assumption:

① According to the given documents, I obtain the size of ~~att~~each attributes which are:

Student:

~~Student~~ Attribute	Type	Size (Byte)
StudentId	Int	4B
Name	Char(15)	15B
NoOfPts	SmallInt	2B
Tutor	Int	4B

1 student tuple = r(S)

= 4+15+2+4

= 25B

Entire: 25 × 50,000 = 1,250,000B = S(S)

Enrolled:

StudentId	Int	4B
CourseId	Char(15)	15B
Term	smallint	2B
Grade	Char(15)	15B

1 tuple = 36B = r(E)

Entire = 36 × 400,000

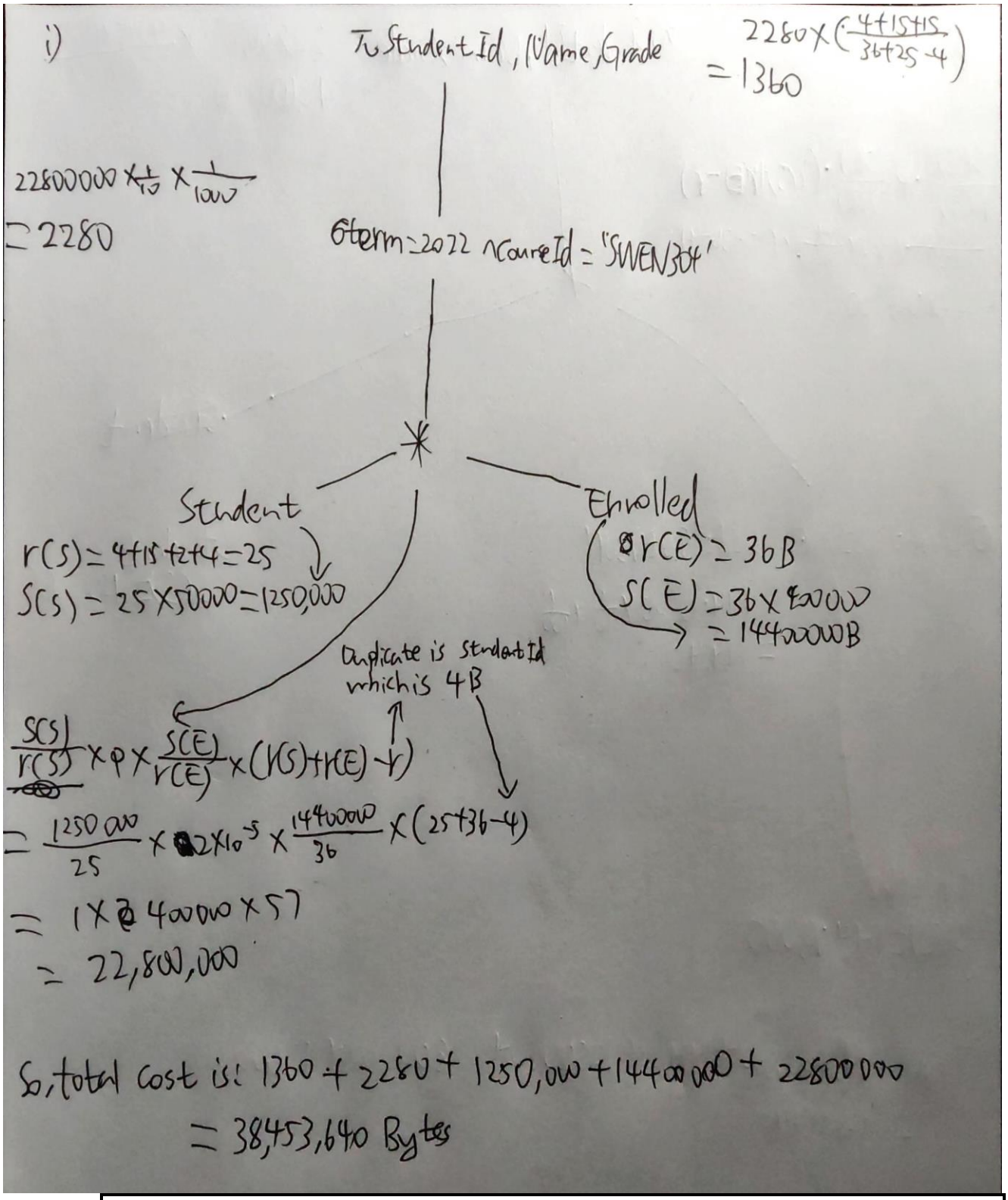
= 14,400,000 B = S(E)

② Then, ~~since~~ there's a Join Node, so the function $\frac{S1}{R1} \cdot p \cdot \frac{S2}{R2} \cdot (R1+R2-r)$ need to be used, where p is the matching Prob that is Not given. So, we assume that the matching Prob can be calculated as $\frac{400,000}{50,000} \times \frac{1}{400,000} = 2 \times 10^{-5} = 0.00002$

And $\frac{8}{400,000} = 2 \times 10^{-5} = p$, which is 8 courses taken by per student.

~~then~~ Then, for the selected node, the probability ~~of term = 2022 ∧ CourseId = 'SWEN304'~~ of term = 2022 ∧ Course Id = 'SWEN304' should be:

$\frac{1}{1000 \text{ courses}} \times \frac{1}{10 \text{ years of students}} = 0.0001 = 1 \times 10^{-4}$

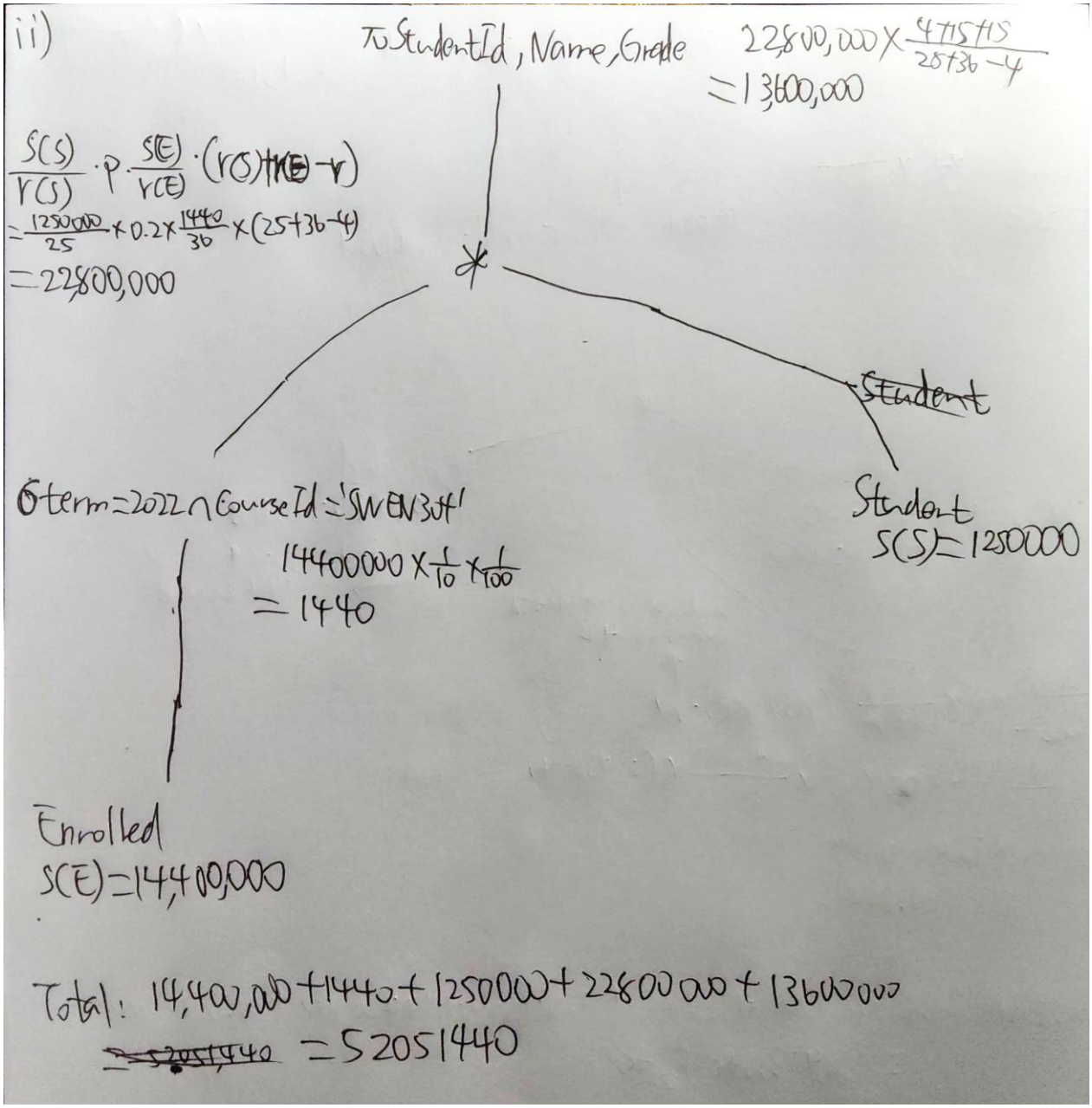

i)
π StudentId, Name, Grade
$2280 \times \left(\frac{4+15+15}{36+25-4}\right) = 1360$
$22800000 \times \frac{1}{10} \times \frac{1}{1000} = 2280$
σTerm=2022 $\wedge$ CourseId = 'SWEN304'
*
Student
$r(S) = 4+15+2+4 = 25$
$S(S) = 25 \times 50000 = 1250,000$
Enrolled
$r(E) = 36B$
$S(E) = 36 \times 40000$
$= 1440000B$
Duplicate is StudentId which is 4B
$\frac{S(S)}{r(S)} \times \rho \times \frac{S(E)}{r(E)} \times (r(S)+r(E)-4)$
$= \frac{1250000}{25} \times 2 \times 10^{-5} \times \frac{1440000}{36} \times (25+36-4)$
$= 1 \times 2 \times 40000 \times 57$
$= 22,800,000$
So, total cost is: 1360 + 2280 + 1250,000 + 1440000 + 22800000
$= 38,453,640$ Bytes

(ii) $\pi_{\text{StudentId, Name, Grade}}$ (Student $*$ $\sigma_{\text{term = 2022} \wedge \text{CourseId = 'SWEN304'}}$ (Enrolled))

ANSWER

The total cost is: 52,051,440 bytes

Assumptions:
1. We can still use some data from previous question, such as the attriute size
2. The probability of Term=2022 /\ CourseId = 'SWEN304' = 0.0001, it is also calculated from previous question.
3. There is also a JOIN node within the query like previous question, so we'll also use the function: s1/r1*P*s2/r2* (r1+r2-r). Since 8 courses are taken by each student (i.e. 400000/50000=8), and for the selected node, the probability will be 1/1000 *1/10 = 0.0001 where 1000 represent courses and 10 represent 10 years of students.
Therefore, we can get that: **P** = 8/(400000*0.0001) = **0.2**



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

ANSWER

Both query trees compute the same query result, therefore the size of the result is the same.
Through the calculation, the first one is smaller, but I think the second one should be smaller, there might be some incorrect calculation.
For the reason why the second tree is smaller, it is because for the 1st one, the Student and Enrolled table is joined first, which need lots of cost, but for the 2nd tree, the Enrolled and Student is joined after the selection is completed which saves the cost compare to the 1st one on the Join node.

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 [20 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.

```
barretts% psql -d a2_giant
psql (10.19)
Type "help" for help.

a2_giant=> VACUUM ANALYZE customer;
VACUUM
a2_giant=>
zsh: suspended  psql -d a2_giant
barretts% psql -d a2_lib
psql (10.19)
Type "help" for help.

a2_lib=> VACUUM ANALYZE customer;
VACUUM
a2_lib=> VACUUM ANALYZE loaned_book;
VACUUM
a2_lib=>
```

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

First, the initial cost is 115.36, which is shown below.

```
a2_giant=> EXPLAIN ANALYZE select count(*) from customer where no_borrowed = 4;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=115.36..115.37 rows=1 width=8) (actual time=1.322..1.326 rows=1 loops=1)
   ->  Seq Scan on customer  (cost=0.00..114.25 rows=443 width=0) (actual time=0.007..0.886 rows=443 loops=1)
         Filter: (no_borrowed = 4)
         Rows Removed by Filter: 4537
 Planning time: 0.056 ms
 Execution time: 1.365 ms
(6 rows)
```

Then, by looking for online tutorial, I create the index on the attribute of no_borrowed in customer table:

```
a2_giant=> CREATE INDEX idx_customer_no_borrowed ON customer(no_borrowed);
CREATE INDEX
```

Then, I execute the query again for analyzing, we can see that the final cost is now about 17.15, by using the formular (original_cost – new_cost) / original_cost, we can get that: (115.36-17.15) / 115.36 = 0.851334951

It shows that it decrease about 85% of the cost

```
a2_giant=> EXPLAIN ANALYZE select count(*) from customer where no_borrowed = 4;
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
-----------------------------------------------
 Aggregate  (cost=17.14..17.15 rows=1 width=8) (actual time=0.991..0.995 rows=1 loops=1)
   ->  Index Only Scan using idx_customer_no_borrowed on customer  (cost=0.28..16.04 rows=443
width=0) (actual time=0.070..0.552 rows=443 loops=1)
         Index Cond: (no_borrowed = 4)
         Heap Fetches: 0
 Planning time: 0.292 ms
 Execution time: 1.044 ms
(6 rows)
```

It can reduce the cost is because before creating index, the database engine have to scan the whole customer table to look for the specificied customer due to there is no index available for the no_borrowed column. It is called sequential scan in which the data engine have to go over all entries until it find the specificied one.

After creating the index, the database engine uses the index for lookup instead of the sequential search, which means it speed up the data retrieval on the table, but as the cost, the additional writes and storage are needed to maintain it. It is actually a kind of trade the space for the time.

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

```
select * from customer where customerid = 4567;
```

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.

ANSWER

Same as previous question, analyse the query first,and we can get that the initial cost is 114.25, which is shown below.

```
a2_giant=> EXPLAIN ANALYZE select * from customer where customerid = 4567;
                                                 QUERY PLAN

-------------------------------------------------------------------------------------------------------
---
 Seq Scan on customer  (cost=0.00..114.25 rows=1 width=56) (actual time=0.809..0.899 rows=1 loops=
1)
   Filter: (customerid = 4567)
   Rows Removed by Filter: 4979
 Planning time: 0.725 ms
 Execution time: 1.000 ms
(5 rows)
```

Then, by looking for online tutorial, I create the index on the attribute of customer_id in customer table:

```
a2_giant=> CREATE INDEX idx_customer_cid ON customer(customerid);
CREATE INDEX
```

Then, I execute the query again for analyzing, we can see that the final cost is now about 8.30, by using the formular (original_cost – new_cost) / original_cost, we can get that:
(114.25-8.30) / 114.25 = 0.927352298
It shows that it decrease about 92.7% of the cost

```
a2_giant=> CREATE INDEX idx_customer_cid ON customer(customerid);
CREATE INDEX
a2_giant=> EXPLAIN ANALYZE select * from customer where customerid = 4567;
                                                        QUERY PLAN

-------------------------------------------------------------------------------------------
-----------------------------------------
 Index Scan using idx_customer_cid on customer  (cost=0.28..8.30 rows=1 width=56)
(actual time=0.052..0.054 rows=1 loops=1)
   Index Cond: (customerid = 4567)
 Planning time: 0.320 ms
 Execution time: 0.104 ms
(4 rows)
```


```
a2_giant=> CREATE INDEX idx_customer_cid ON customer(customerid);
CREATE INDEX
a2_giant=> EXPLAIN ANALYZE select * from customer where customerid = 4567;
                                                QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------
 Index Scan using idx_customer_cid on customer  (cost=0.28..8.30 rows=1 width=56) (actual time=0.052..0.054 rows=1 loops=1)
   Index Cond: (customerid = 4567)
 Planning time: 0.320 ms
 Execution time: 0.104 ms
(4 rows)
```


The reason is actually the same as the previous question since I use the same method, which is creating the index.
It can reduce the cost is because before creating index, the database engine have to scan the whole customer table to look for the specificied customer due to there is no index available for the customer_id column. It is called sequential scan in which the data engine have to go over all entries until it find the specificied one.
After creating the index, the database engine uses the index for lookup instead of the sequential search, which means it speed up the data retrieval on the table, but as the cost, the additional writes and storage are needed to maintain it. It is actually a kind of trade the space for the time.

c) [10 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

First,lets us see the output of the given query.

```
a2_lib=> select clb.f_name, clb.l_name, noofbooks
a2_lib-> from (select f_name, l_name, count(*) as noofbooks
a2_lib(>      from customer natural join loaned_book
a2_lib(>      group by f_name, l_name) as clb
a2_lib->      where 3 > (select count(*)
a2_lib(>                 from (select f_name, l_name, count(*) as noofbooks
a2_lib(>                       from customer natural join loaned_book
a2_lib(>                       group by f_name, l_name) as clb1
a2_lib(>                       where clb.noofbooks<clb1.noofbooks)
a2_lib->                       order by noofbooks desc;
     f_name      |     l_name      | noofbooks
-----------------+-----------------+-----------
 Thomson         | Wayne           |         5
 May-N           | Leow            |         4
 Peter           | Andreae         |         3
(3 rows)
```

And analyse the query,and from below screenshot, we can get that the initial cost is 3.05..85.89,which choose the upper bound, that is 85.59.

```
a2_lib=> EXPLAIN ANALYSE select clb.f_name, clb.l_name, noofbooks
a2_lib-> from (select f_name, l_name, count(*) as noofbooks
a2_lib(>      from customer natural join loaned_book
a2_lib(>      group by f_name, l_name) as clb
a2_lib->      where 3 > (select count(*)
a2_lib(>                 from (select f_name, l_name, count(*) as noofbooks
a2_lib(>                       from customer natural join loaned_book
a2_lib(>                       group by f_name, l_name) as clb1
a2_lib(>                       where clb.noofbooks<clb1.noofbooks)
a2_lib->                       order by noofbooks desc;
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=86.01..86.03 rows=8 width=40) (actual time=0.864..0.890 rows=3 loops=1)
   Sort Key: clb.noofbooks DESC
   Sort Method: quicksort  Memory: 25kB
   ->  Subquery Scan on clb  (cost=3.05..85.89 rows=8 width=40) (actual time=0.661..0.834 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=0.276..0.302 rows=15 loops=1)
               Group Key: customer.f_name, customer.l_name
               ->  Hash Join  (cost=1.52..2.86 rows=26 width=32) (actual time=0.147..0.235 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=0.017..0.043 rows=26 loops=1)
                     ->  Hash  (cost=1.23..1.23 rows=23 width=36) (actual time=0.080..0.083 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=0.004..0.040 rows=23 loops=1)
         SubPlan 1
           ->  Aggregate  (cost=3.57..3.58 rows=1 width=8) (actual time=0.030..0.031 rows=1 loops=15)
                 ->  HashAggregate  (cost=3.05..3.28 rows=23 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(*))
                       Rows Removed by Filter: 11
                       ->  Hash Join  (cost=1.52..2.86 rows=26 width=32) (actual time=0.138..0.225 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=0.003..0.032 rows=26 loops=1)
                             ->  Hash  (cost=1.23..1.23 rows=23 width=36) (actual
time=0.094..0.097 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=0.004..0.056 rows=23 loops=1)
 Planning time: 1.370 ms
 Execution time: 1.232 ms
(28 rows)
```

For improving, we need to analyse the given query.I find out that there is duplicate Common Table Expressions(CTE) in the final where clause,the CTE that start as select count(*) is unnecessary. All we need is to find whom the noofbooks is equal or more than 3, which means we can straightly use the clb in where clause. Therefore, my modified optimized query is shown below, we can clearly see that the output is still these 3 rows which is the same as the original.

```
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
        order by noofbooks desc
    ) as clb
WHERE clb.noofbooks >= 3;
```

```
a2_lib=> select clb.f_name,
a2_lib->     clb.l_name,
a2_lib->     noofbooks
a2_lib-> from (
a2_lib(>         select f_name,
a2_lib(>             l_name,
a2_lib(>             count(*) as noofbooks
a2_lib(>         from customer
a2_lib(>             natural join loaned_book
a2_lib(>         group by f_name,
a2_lib(>             l_name
a2_lib(>         order by noofbooks desc
a2_lib(>     ) as clb
a2_lib-> WHERE clb.noofbooks >= 3;
     f_name      |     l_name      | noofbooks
-----------------+-----------------+-----------
 Thomson         | Wayne           |         5
 May-N           | Leow            |         4
 Peter           | Andreae         |         3
(3 rows)
```

Then, I call "EXPLAIN ANALYZE" on my query for analyzing, we can see that for now, the final cost is 3.35, by using the formular:
(original_cost - new_cost) / original_cost, we can get that:
(85.59-3.35) / 85.59 = 0.960859914
It shows that it decrease about 96% of the cost

```
a2_lib=> EXPLAIN ANALYSE select clb.f_name,
a2_lib->     clb.l_name,
a2_lib->     noofbooks
a2_lib-> from (
a2_lib(>         select f_name,
a2_lib(>             l_name,
a2_lib(>             count(*) as noofbooks
a2_lib(>         from customer
a2_lib(>             natural join loaned_book
a2_lib(>         group by f_name,
a2_lib(>             l_name
a2_lib(>         order by noofbooks desc
a2_lib(>     ) as clb
a2_lib-> WHERE clb.noofbooks >= 3;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=3.87..3.92 rows=23 width=40) (actual time=0.252..0.266 rows=3 loops=1)
   Sort Key: (count(*)) DESC
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (cost=3.12..3.35 rows=23 width=40) (actual time=0.221..0.234 rows=3 loops=1)
         Group Key: customer.f_name, customer.l_name
         Filter: (count(*) >= 3)
         Rows Removed by Filter: 12
         ->  Hash Join  (cost=1.52..2.86 rows=26 width=32) (actual time=0.094..0.183 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=0.006..0.031 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  (cost=0.00..1.23 rows=23 width=36) (actual time=0.005..0.032 rows=23 loops=1)
 Planning time: 0.235 ms
 Execution time: 0.329 ms
(15 rows)
```

As I've mentioned above, the common table expression in where clause is completely unnecessary,it can only increase the cost. So, I modify a little bit on that, and change the where condition, we can get the same result with lower cost, which means it is more efficient.

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

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 comp302tools

or

> need postgresql

You may wish to add either "`need comp302tools`", 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!