Student ID: 300476924 **First Name: Zichu** **Last Name: Yan**

School of Engineering and Computer Science

SWEN 304 **Database System Engineering**

Assignment 2

Due: 23:59, Monday, 16 September 2019

The objective of this assignment is to test your understanding of Relational Algebra and Query Processing and Optimization. It is worth 5.0% 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 assignment in **pdf** with your **student ID and Name** via the submission system.
- Submissions not in pdf, or without ID and name will incur **3 marks** deduction from the total marks.

Question 1. Relational Algebra [20 marks]

Consider the Suppliers database schema given below.

Set of relation schemas:

Products ({*PId*, *Description, Category*}, {*PId*}),
Manufacturer ({*MId*, *Name*, *Phone*}, {*MId*})
Produced_By ({*PId*, *MId, Amount*}, {*PId* + *MId*})

Set of referential integrity constraints:
Produced_By [*PId*] ⊆ *Products* [*PId*],
Produced_By [*MId*] ⊆ *Manufacturer* [*MId*]

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

1) Retrieve the names of all manufacturers who produce some products of category food or 'healthcare'.

$\pi_{name}((\sigma_{Category='food' \ V \ 'healthcare'}(r(Products))) * r(Produced\_By) * r(Manufacturer))$

2) Retrieve the names of all manufacturers who always produce products of category 'drink'.

$\pi_{name}((\sigma_{Category='drink'}(r(Products))) * r(Produced\_By) * r(Manufacturer)) -$

$\pi_{name}((\sigma_{Category\ != \ 'drink'}(r(Products))) * r(Manufacturer) * r(Produced\_By))$

3) Retrieve the descriptions of all products that are produced by two or more manufacturers.

$\pi_{Description}(\pi_{PID}(\sigma_{MID\ != MIDo}(\sigma_{MID}\text{->MIDo}\ (\pi_{PID,MID}(r(Produced\_By)) * (\pi_{PID,MID}(r(Produced\_By))) * r(Product)$

4) For all products of category 'food' list their descriptions and the names of their manufacturers.

$\pi_{Description,\ Name}((\sigma_{Category='food'}(r(Products))) * r(Produced\_By) * r(Manufacturer))$

b) [8 marks] Translate the following two queries <u>into plain English</u> ***and*** <u>into SQL</u>:

1) $\pi_{Phone}(r(Products) * (\sigma_{Amount>50}(r(Produced\_By)) * r(Manufacturer))$

Retrieve the phonenumber of all manufacturers whose products with a yield is greater than 50.

```
SELECT Phone
FROM Products NATURAL JOIN Produced_By NATURAL JOIN Manufacturer
WHERE Amount>50;
```

2) $\pi_{MId}(\sigma_{Amount>50}(r(Produced\_By))) \cap \pi_{MId}(r(Produced\_By) * (\sigma_{Description='Muffin'}(r(Products))))$

Retrieve the MID of all products which production is greater than 50 and description is 'Muffin'.

```
SELECT MID
FROM Produced_By NATURAL JOIN Products
WHERE Amount>50 AND Description ='Muffin';
```

Question 2. Heuristic and Cost-Based Query Optimization [50 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**

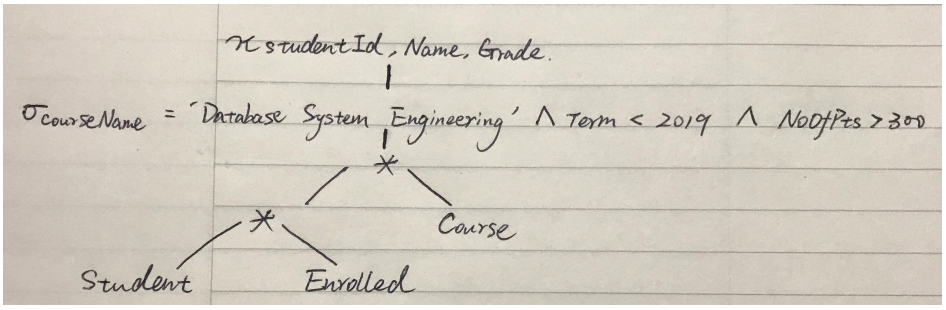
Suppose we are given a query in SQL

```
SELECT StudentId, Name, Grade

FROM Student NATURAL JOIN Enrolled NATURAL JOIN Course

WHERE CourName = 'Database Systems Engineering' AND Term < 2019
                AND NoOfPts > 300;
```

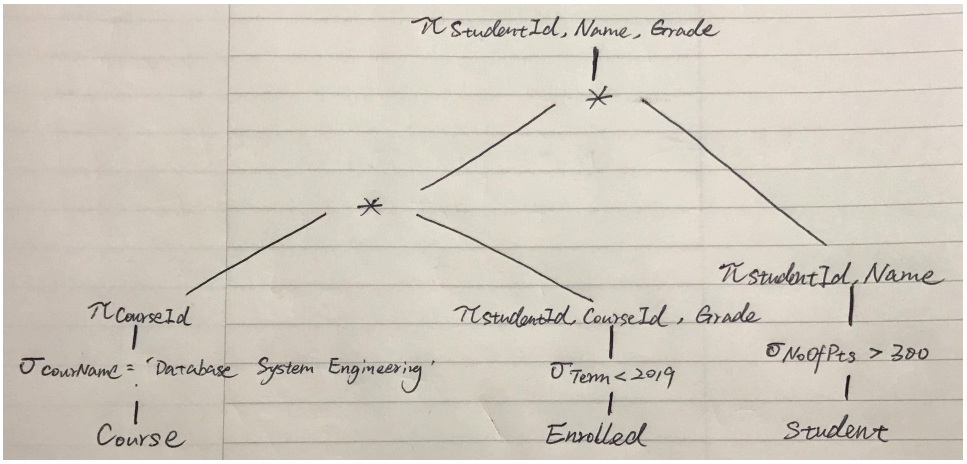
1) **[5 marks]** Transfer the above given query into Relational Algebra.

$\pi_{StudentId,Name,Grade}$ $((\sigma_{NoOfPts>300}(r(Student)))$ * $(\sigma_{Term<2019}(r(Enrolled)))$ * $(\sigma_{CourName='Database\ System\ Engineering'}(r((Course)))$

2) **[5 marks]** Draw a query tree corresponding to your answer for subquestion **1)**.



3) **[10 marks]** Transfer the query tree from **2)** into an optimized query tree using the query optimization heuristics.



b) **[30 marks] Query cost calculation**

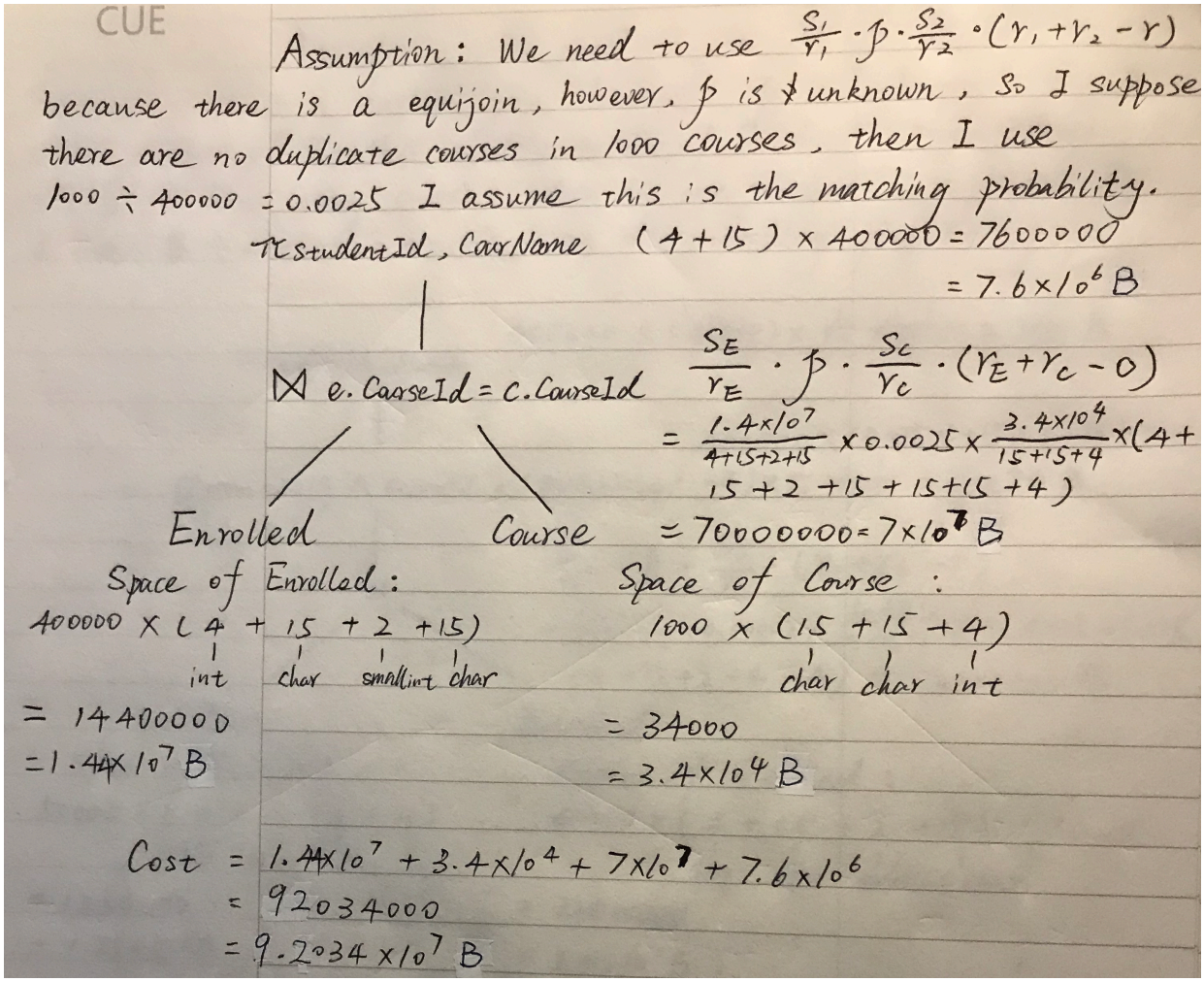
Suppose the following:

- 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* $n_e = 400000$ *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.

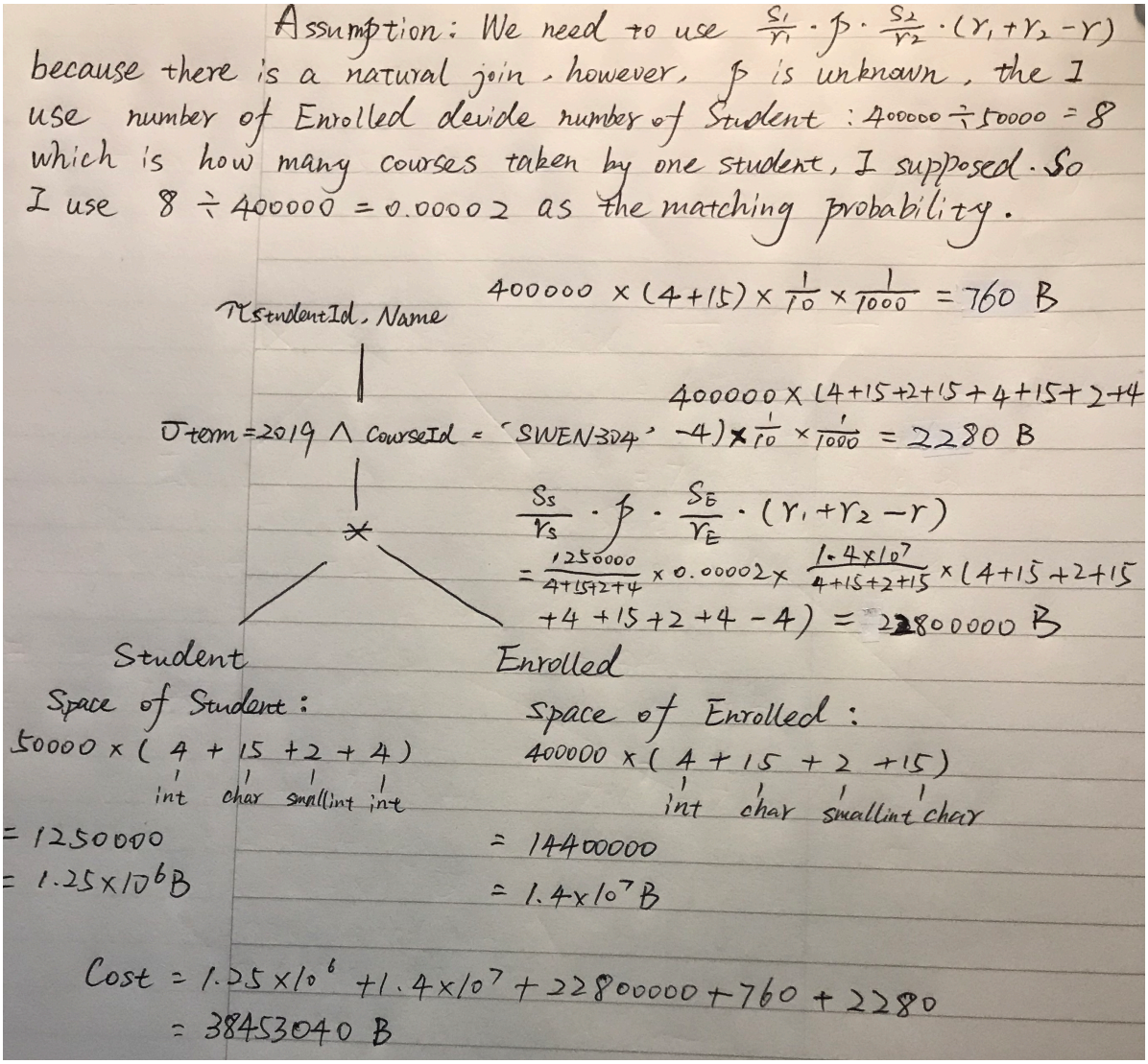
1) [10 marks] For the given query below draw a query tree and calculate the cost of executing the query.

$\pi_{\text{StudentId, CourName}}$ (r(Enrolled) $\bowtie_{\text{e.CourseId = c.CourseId}}$ r(Course))



2) **[20 marks]** For the given query below draw a query tree and calculate the cost of executing query.

$$\pi_{\texttt{StudentId, Name}}\ (\sigma_{\text{term}=2019 \wedge \text{CourseId}='\text{SWEN304}'}(r(\text{Student}) \ * \ r(\text{Enrolled})))$$



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 [30 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) [10 marks] Improve the cost estimate of the following query:

```
select count(*) from customer where no_borrowed = 6;
```

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.

```
[YanzichA2=> EXPLAIN ANALYZE select count(*) from customer where no_borrowed = 6;
                                                          QUERY PLAN
------------------------------------------------------------------------------
------------------------------
 Aggregate  (cost=114.41..114.42 rows=1 width=8) (actual time=333.870..333.892 r
ows=1 loops=1)
   ->  Seq Scan on customer  (cost=0.00..114.25 rows=63 width=0) (actual time=14
.659..332.955 rows=63 loops=1)
         Filter: (no_borrowed = 6)
         Rows Removed by Filter: 4917
 Planning time: 0.148 ms
 Execution time: 334.127 ms
(6 rows)
```

Before I create index the result is 114.41 which cost too much.

```
[YanzichA2=> Create index customer_books on customer(no_borrowed);
 CREATE INDEX
[YanzichA2=> EXPLAIN ANALYZE select count(*) from customer where no_borrowed = 6;
                                                          QUERY PLAN
------------------------------------------------------------------------------
------------------------------
 Aggregate  (cost=5.54..5.55 rows=1 width=8) (actual time=10.974..10.996 rows=1 loops=1)
   ->  Index Only Scan using customer_books on customer  (cost=0.28..5.38 rows=63 width=0) (actual time=9.497..10.2
42 rows=63 loops=1)
         Index Cond: (no_borrowed = 6)
         Heap Fetches: 0
 Planning time: 1.608 ms
 Execution time: 11.259 ms
(6 rows)
```

After I create the index the result become much better , it become 5.55 which decease dramatically from 114.42.The cost decrease 95% . I created index for no_borrowed which can speed up data retrieval. Because searching index is more efficient than searching sequences . “ Index all the predicates in JOIN, WHERE, ORDER BY and GROUP BY clauses. WebSphere Commerce typically depends heavily on indexes to improve SQL performance and scalability. Without proper indexes, SQL queries can cause table scans, which causes either performance or locking problems. It is recommended that all predicate columns be indexed. The exception being where column data has very low cardinality.” -https://www.ibm.com/support/knowledgecenter/en/SSZLC2_9.0.0/com.ibm.commerce.developer.doc/refs/rsdperformanceworkspaces.htm

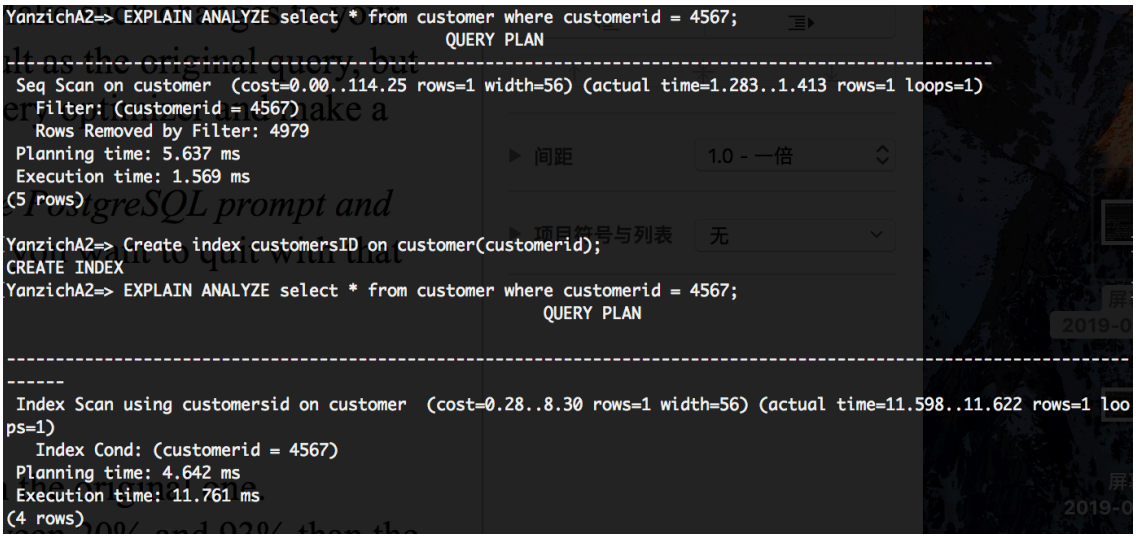
Marking schedule:
You will receive:

- 7 marks if your query cost estimate is at least 64% better than the original one.
- between 3 and 7 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 3 additional marks if you give reasonable explanations of what you have done.

b) [7 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.



Before using index the cost is 114.25 ,after using index the cost decrease a lot 8.30.The cost decrease 90%. Actually it is same reason as last question. I created index for customerid which can speed up data retrieval because searching index is more efficient than searching sequences . " Index all the predicates in JOIN, WHERE, ORDER BY and GROUP BY clauses. WebSphere Commerce typically depends heavily on indexes to improve SQL performance and scalability. Without proper indexes, SQL queries can cause table scans, which causes either performance or locking problems. It is recommended that all predicate columns be indexed. The exception being where column data has very low cardinality."

Marking schedule:
You will receive

- 5 marks if your query cost estimate is 93% (or more) better than the original one.
- between 1 and 5 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 2 additional marks if you give reasonable explanations of what you have done.

c) [13 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:

```
Explain analyze 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.

This is the given query:

```
YanzichA22=> select clb.f_name, clb.l_name, noofbooks
YanzichA22-> from (select f_name, l_name, count(*) as noofbooks
YanzichA22(>       from customer natural join loaned_book
YanzichA22(>       group by f_name, l_name) as clb
YanzichA22->       where 3 > (select count(*)
YanzichA22(>                  from (select f_name, l_name, count(*) as noofbooks

YanzichA22(>                        from customer natural join loaned_book
YanzichA22(>                        group by f_name, l_name) as clb1
YanzichA22(>                        where clb.noofbooks<clb1.noofbooks)
YanzichA22->                        order by noofbooks desc;
 f_name  |  l_name  | noofbooks
---------+----------+-----------
 Thomson | Wayne    |         5
 May-N   | Leow     |         4
 Peter   | Andreae  |         3
(3 rows)
```

When I do EXPLAIN ANALYZE I got this result:

```
YanzichA22=> Explain analyze select clb.f_name, clb.l_name, noofbooks
YanzichA22-> from (select f_name, l_name, count(*) as noofbooks
YanzichA22(>       from customer natural join loaned_book
YanzichA22(>       group by f_name, l_name) as clb
YanzichA22->       where 3 > (select count(*)
YanzichA22(>                  from (select f_name, l_name, count(*) as noofbooks
YanzichA22(>                        from customer natural join loaned_book
YanzichA22(>                        group by f_name, l_name) as clb1
YanzichA22(>                        where clb.noofbooks<clb1.noofbooks)
YanzichA22->                        order by noofbooks desc;
                                                                  QUERY PLAN

------------------------------------------------------------------------------------------
-----------------------------------------------------------
 Sort  (cost=86.01..86.03 rows=8 width=46) (actual time=7.327..7.371 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=46) (actual time=5.596..7.174 ro
ws=3 loops=1)
         Filter: (3 > (SubPlan 1))
         Rows Removed by Filter: 12
         ->  HashAggregate  (cost=3.05..3.28 rows=23 width=46) (actual time=2.122..2.309 row
s=15 loops=1)
               Group Key: customer.f_name, customer.l_name
               ->  Hash Join  (cost=1.52..2.86 rows=26 width=38) (actual time=0.864..1.769 r
ows=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=0.036..0.341 rows=26 loops=1)
                     ->  Hash  (cost=1.23..1.23 rows=23 width=42) (actual time=0.655..0.666
rows=23 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 10kB
                           ->  Seq Scan on customer  (cost=0.00..1.23 rows=23 width=42) (act
ual time=0.036..0.320 rows=23 loops=1)
         SubPlan 1
           ->  Aggregate  (cost=3.57..3.58 rows=1 width=8) (actual time=0.274..0.285 rows=1
loops=15)
                 ->  HashAggregate  (cost=3.05..3.28 rows=23 width=46) (actual time=0.157..0.206 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=38) (actual time=0.889..1.791 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=0.031..0.336 rows=26 loops=1)
                             ->  Hash  (cost=1.23..1.23 rows=23 width=42) (actual time=0.647..0.658 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=42) (actual time=0.032..0.315 rows=23 loops=1)
 Planning time: 1.526 ms
 Execution time: 8.209 ms
(28 rows)
```

What we want is find the f_name, l_name and number of books who borrowed more than 3.
Then I simplified the query to make it cheaper by remove the redundant count(*):
(Explain analyze)
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;

```
YanzichA22=> SELECT clb.f_name, clb.l_name, noofbooks
YanzichA22-> From (SELECT f_name, l_name, count(*) as noofbooks
YanzichA22(> From customer natural join loaned_book
YanzichA22(> group by f_name, l_name
YanzichA22(> order by noofbooks DESC) as clb;
     f_name      |     l_name      | noofbooks
-----------------+-----------------+-----------
 Thomson         | Wayne           |         5
 May-N           | Leow            |         4
 Peter           | Andreae         |         3
 James           | Noble           |         2
 Ewan            | Tempero         |         2
 Nigel           | Somerfield      |         1
 Linda           | McMurray        |         1
 Kirk            | Jackson         |         1
 Gang            | Xu              |         1
 Jerome          | Dolman          |         1
 Craig           | Anslow          |         1
 Daisy           | Zhou            |         1
 Customer        | Default         |         1
 Chang           | Chui            |         1
 Shusen          | Yi              |         1
(15 rows)
```

We can see top3 tuples are same as the original query ,then we can check the cost:

```
YanzichA22=> Explain analyze SELECT clb.f_name, clb.l_name, noofbooks
YanzichA22-> From (SELECT f_name, l_name, count(*) as noofbooks
YanzichA22(> From customer natural join loaned_book
YanzichA22(> group by f_name, l_name
YanzichA22(> order by noofbooks DESC) as clb;
                                                                  QUERY PLAN

------------------------------------------------------------------------------
-----------------------------------------
 Sort  (cost=3.80..3.86 rows=23 width=46) (actual time=2.460..2.636 rows=15 loop
s=1)
   Sort Key: (count(*)) DESC
   Sort Method: quicksort  Memory: 26kB
   ->  HashAggregate  (cost=3.05..3.28 rows=23 width=46) (actual time=2.020..2.2
07 rows=15 loops=1)
         Group Key: customer.f_name, customer.l_name
         ->  Hash Join  (cost=1.52..2.86 rows=26 width=38) (actual time=0.742..1
.665 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) (a
ctual time=0.038..0.357 rows=26 loops=1)
               ->  Hash  (cost=1.23..1.23 rows=23 width=42) (actual time=0.611..
0.622 rows=23 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 10kB
                     ->  Seq Scan on customer  (cost=0.00..1.23 rows=23 width=42
) (actual time=0.036..0.319 rows=23 loops=1)
 Planning time: 0.906 ms
 Execution time: 3.190 ms
(13 rows)
```

The cost become much cheaper and I think it is because original query used too many count(*) which is expensive operation. So I remove the redundant one which make the result get better from 86.03 to 3.86, get 95% better.

Marking schedule:

You will receive:

- 3 marks if you explain in English how the query computes the answer,
- 7 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 3 marks if you give reasonable explanations of why the query given is inefficient and why is your query better.

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

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, so you need to run it from a Unix prompt in a shell window. To enable the various applications required, first 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!