SWEN304
Assignment two

Siwen Feng
300363512

Question 1:

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'.

πname (σcategory ='food' ∨ σcategory = healthcare'((r(Products)*r(Produced_by))*(r(Manufacturer)))

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

πname (σcategory ='drink'((r(Products)*r(Produced_by))*(r(Manufacturer))) - πname (σcategory <>'drink'((r(Products)*r(Produced_by))*(r(Manufacturer)))

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

πdescription(πPid(σMid <> σMid_other (δMid->δMid_other(πPid,Mid(r(Produced_by))* (πPid,Mid(r(Produced_by))*r(Products)) Get some help from tutor

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

πdescription,name(σcategory ='food' ((r(Products)*r(Produced_by))*(r(Manufacturer)))

b) [8 marks] Translate the following two queries into plain English and into SQL:

1) πPhone (r(Products) * (σAmount>50 (r(Produced_By))) * r(Manufacturer))

Retrieve phone numbers of the manufacturers who provided more than 50 products.

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

2) πMId (σAmount>50 (r(Produced_By))) ∩ πMId (r(Produced_By) * (σDescription='Muffin' (r(Products))))

Retrieve Mid of manufacturer which produce more than 50 muffins.

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

Question 2:

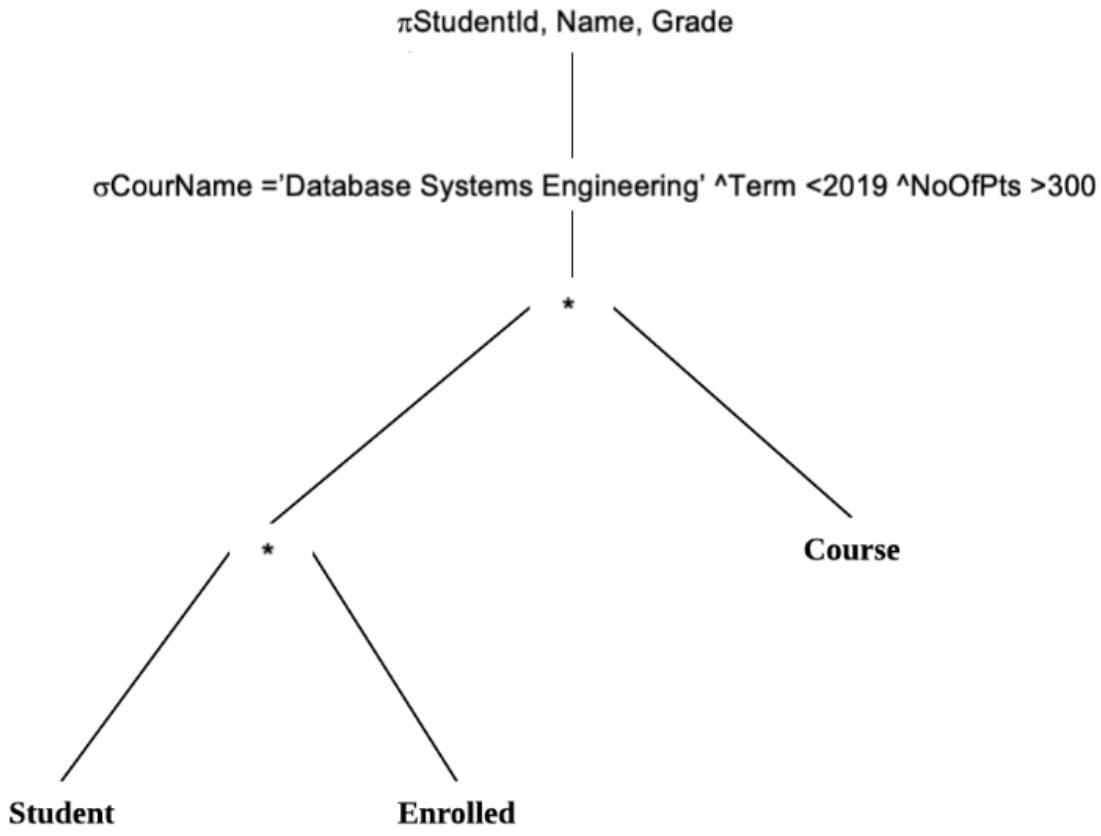
a) [20 marks] Heuristic query optimization Suppose we are given a query in SQL

```
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.

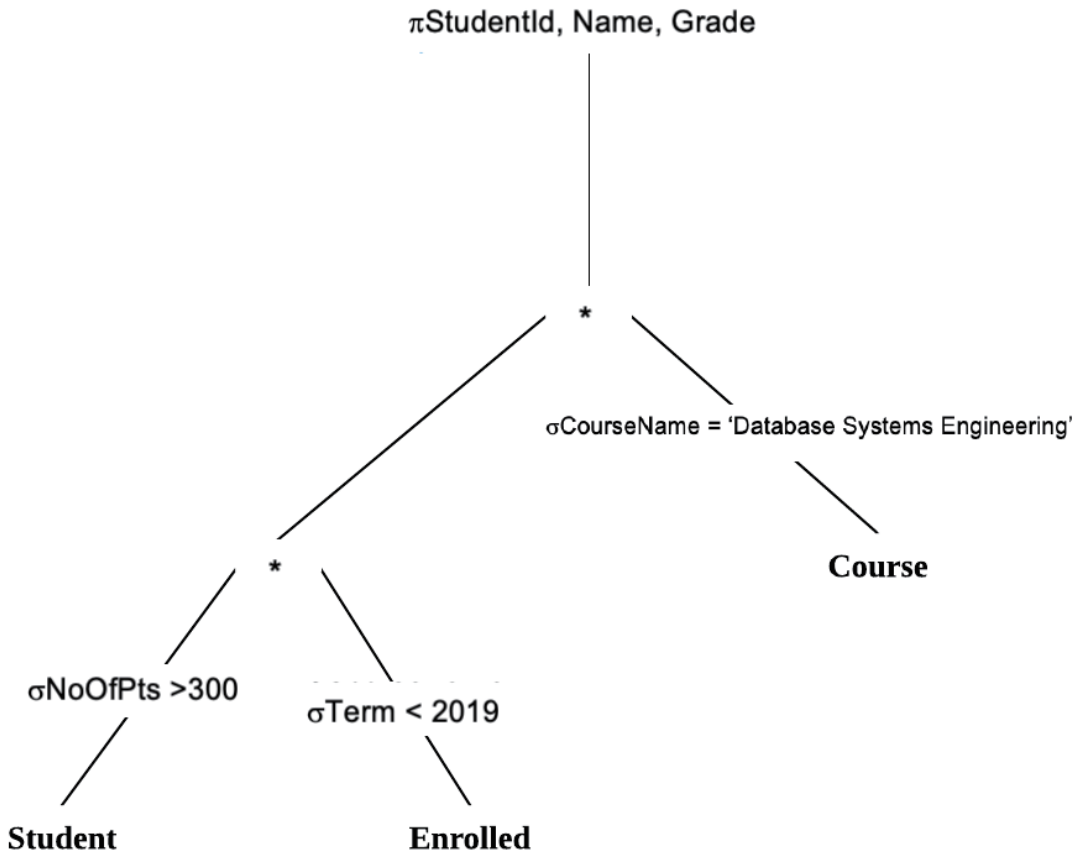
πStudentId,Name,Grade(σCourName ='Database Systems Engineering' ^Term <2019 ^NoOfPts >300((r(Student)*r(Enrolled))*r(Course)))

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

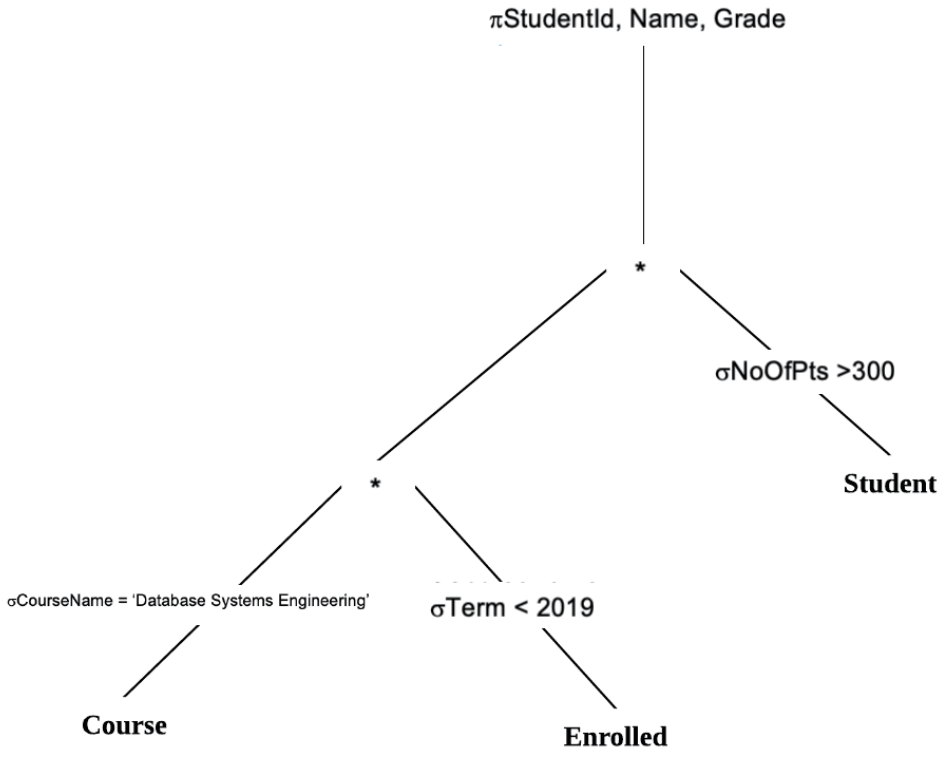


3) [10 marks] Transfer the query tree from 2) into an optimized query tree using the query optimization heuristics.

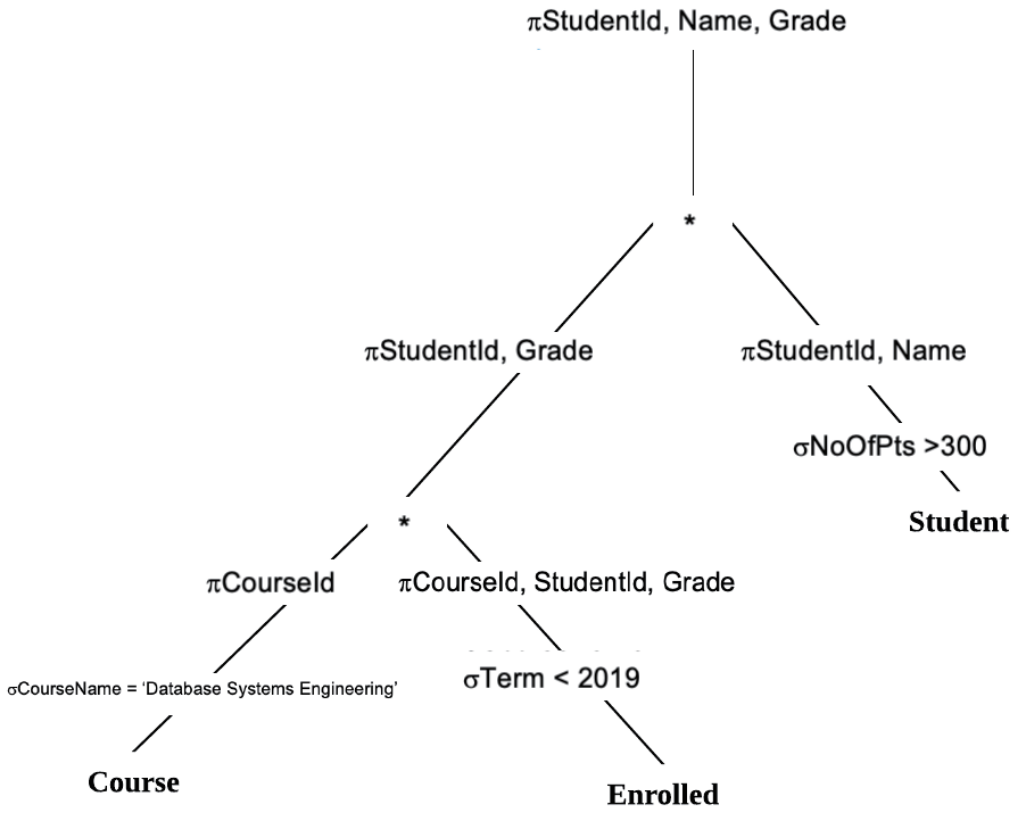
Apply for the rule 6 move select operation down the tree.



Apply for rule 9 Switching join based on frequency, Students who have more than 300 is bigger than courses which name is database systems engineering. Thus, we can swap node student and node course.



Apply for rule 7 to the tree we can get the final tree below:



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

π StudentId, CourName (r(Enrolled) e.CourseId = c.CourseId r(Course))

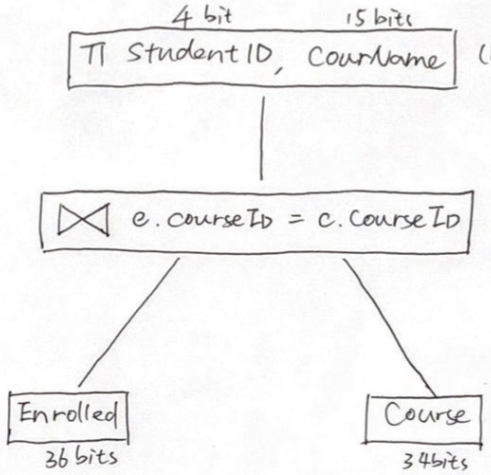
* Assumption: the matching probability of this question should be 1000/400,000. Because, there are 400,000 enrolments, and 1000 different courses. If these two table match with eachother the p should be $1000/400{,}000 = \frac{1}{400}$

$r_e = 4+15+2+15 = 36$ bits

$r_c = 15+15+4 = 34$ bits

$S_e = 36 \times 400000 = 14{,}400{,}000$

$S_c = 34 \times 1000 = 34{,}000$



$\frac{S_e}{r_e} \times p \times \frac{S_c}{r_c} \times (r_e + r_c - r)$

Because it is equal join

$r = 0$

$= 400000 \times \frac{1}{400} \times 1000 \times (34+36-0)$

$= 70{,}000{,}000$

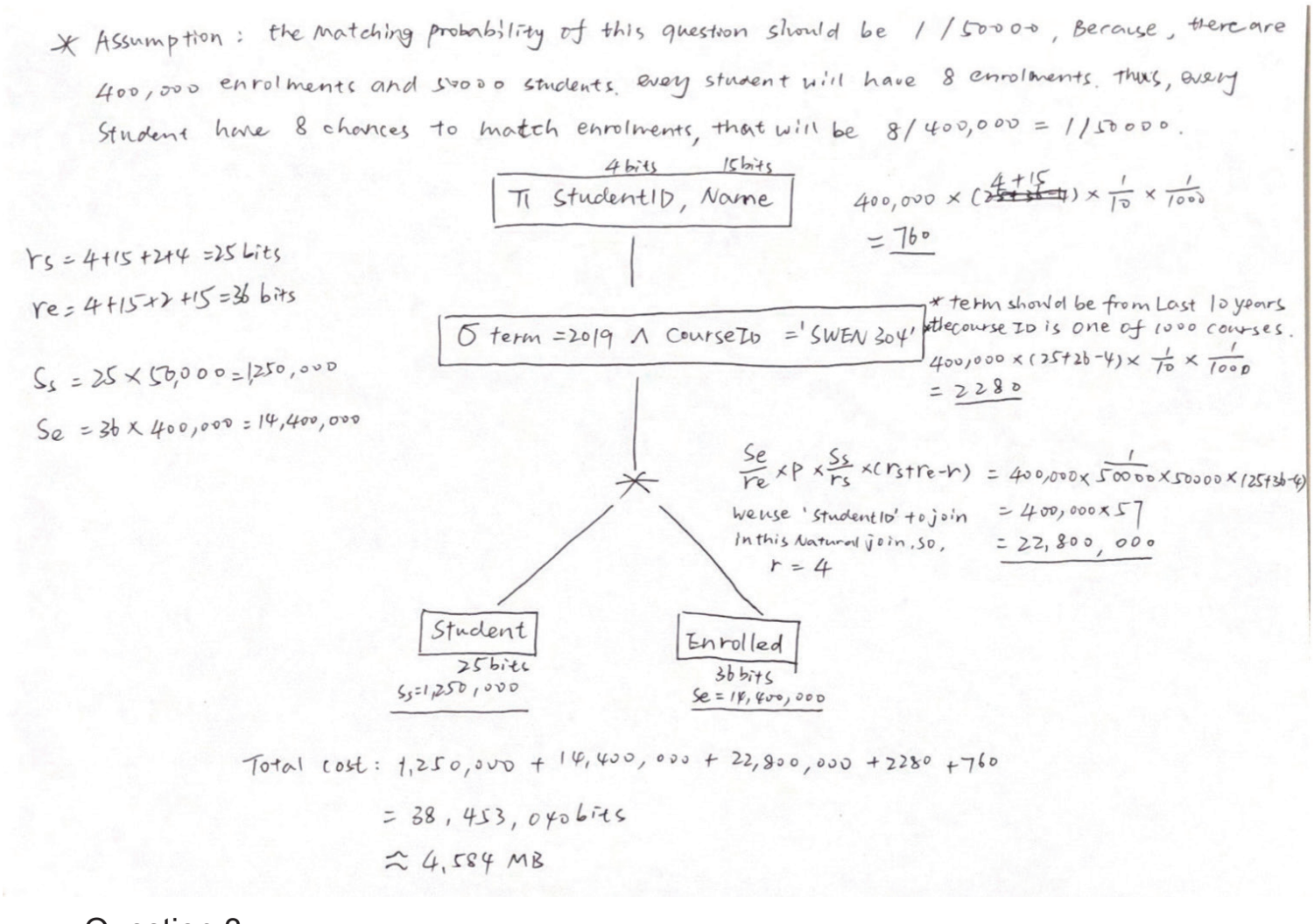
Total Cost : Sum all nodes in the tree : 7,600,000 + 70,000,000 + 14,400,000 + 34,000

$= 92{,}034{,}000$

≈ 10.9713 MB

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

πStudentId, Name (σterm =2019 ∧ CourseId ='SWEN304' (r(Student) * r(Enrolled)))



Question 3:

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

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

```
FAss2G=> EXPLAIN ANALYZE select count(*) from customer where no_borrowed = 6;
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=114.41..114.42 rows=1 width=8) (actual time=2.699..2.721 rows=1 loops=1)
   ->  Seq Scan on customer  (cost=0.00..114.25 rows=63 width=0) (actual time=0.052..1.957 rows=63 loops=1)
         Filter: (no_borrowed = 6)
         Rows Removed by Filter: 4917
 Planning time: 0.188 ms
 Execution time: 2.880 ms
(6 rows)
```

It is the original query which has cost 114.42. We always think the worst situation to improve performance. Although it is a simple query it has a high cost. The reason for it may because this query scanned whole customer table. For optimizing the cost of the query, there are multiple ways to achieve our goals. We can use ‘order by’ to sort the number of borrowed books. The output is down below:

```
FAss2G=> EXPLAIN ANALYZE select count(*) from customer where no_borrowed = 6 GROUP BY no_borrowed
ORDER BY no_borrowed;
                                                    QUERY PLAN

-------------------------------------------------------------------------------------------------------------
----------
 GroupAggregate  (cost=0.00..114.66 rows=10 width=12) (actual time=2.704..2.726 rows=1 loops=1)
   Group Key: no_borrowed
   ->  Seq Scan on customer  (cost=0.00..114.25 rows=63 width=4) (actual time=0.052..1.943 rows=63
 loops=1)
         Filter: (no_borrowed = 6)
         Rows Removed by Filter: 4917
 Planning time: 0.211 ms
 Execution time: 2.882 ms
(7 rows)
```

If we only consider the worst case, the cost even increased after ordering the number of books. It still uses the sequential scan to scan the whole customer table. Because we have a lot of data in this table which is around 5000. Thus, some other scan techniques can be implemented to improve efficiency. There are two scans may occur, index scan and bitmap scan. However, I should create an index for no_borrowed.

```
FAss2G=> CREATE INDEX quick_search_no_borrowed ON customer(no_borrowed);
CREATE INDEX
FAss2G=> EXPLAIN ANALYZE select count(*) from customer where no_borrowed = 6;
                                                                  QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------------
----------------------------
 Aggregate  (cost=5.54..5.55 rows=1 width=8) (actual time=11.907..11.929 rows=1 loops=1)
   ->  Index Only Scan using quick_search_no_borrowed on customer  (cost=0.28..5.38 rows=63 width=0) (actual time=10.4
14..11.163 rows=63 loops=1)
         Index Cond: (no_borrowed = 6)
         Heap Fetches: 0
 Planning time: 2.317 ms
 Execution time: 12.195 ms
(6 rows)
```

After I created the index for this table on no_borrowed. The cost decreased dramatically. From 114.66 to 5.55. Because this query is asking for one value. The system use Index only scan to find out our target. Thus, the performance increased a lot.

The increase is around 114.66-5.55/114/66 = 95.15%.

However, the bitmap scan will use the index to identify the portion and use the sequential scan to find a concrete target. In this scenario, the query subsets are small. The index scan is good enough to deal with this situation.

Explanation: Why index scan has a better performance for this query? Compared with the sequential scan, the advantage of the index scan is very efficient when we only select a small amount of data. In our query, we only selected ‘no_borrowed’ = 6. The results of this datasets are 63 rows. Thus, the index scan is a suitable scan method for our question. However, Sequential scan is good at handling with a selection of large rows.

b) [7 marks] Improve the efficiency of the following query:

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

```
FAss2G=> 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=0.719..0.809 rows=1 loops=1)
   Filter: (customerid = 4567)
   Rows Removed by Filter: 4979
 Planning time: 0.485 ms
 Execution time: 0.908 ms
(5 rows)
```

The original query has cost up to 114.25. It also uses a sequential scan. Thus, to change the sequential scan to index id, I create another index for customer Id.

```
FAss2G=> CREATE INDEX quick_search_customerId ON customer(customerID);
CREATE INDEX
FAss2G=> EXPLAIN ANALYZE select l_name,f_name from customer where customerid = 4567;
                                                        QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------------------------
---
 Index Scan using quick_search_customerid on customer  (cost=0.28..8.30 rows=1 width=32) (actual time=0.196..0.220 rows=1 loops=
1)
   Index Cond: (customerid = 4567)
 Planning time: 1.090 ms
 Execution time: 0.345 ms
(4 rows)
```

After giving the index for customer Id, the performance increased from 114.25 to 8.30. The percentage increased ***114.25-8.30/114.25 = 92.73% It almost increased by 93%***.

Explanation: It is the same reason with the last question. The index scan is more suitable for selecting small dataset. The target is a very specific row when customer Id equal to 4567, there is only one row being selected. Then the index scan is more efficient.

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:

```
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;
```

The original query has poor performance. It contains two main parts. The first part is to select the customer's name and the number of books they borrowed. The second part is used twice count(*) functions to get a customer who has less than 3 people who borrowed books than him/her. Thus, they can find the top three customers who borrowed books more than others.

```
                                                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=86.01..86.03 rows=8 width=46) (actual time=6.869..6.914 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=5.215..6.803 rows=3 loops=1)
         Filter: (3 > (SubPlan 1))
         Rows Removed by Filter: 12
         ->  HashAggregate  (cost=3.05..3.28 rows=23 width=46) (actual time=1.953..2.141 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=0.697..1.600 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=0.035..0.340 rows=26 loops=1)
                     ->  Hash  (cost=1.23..1.23 rows=23 width=42) (actual time=0.612..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=0.035..0.318 rows=23 loops=1)
         SubPlan 1
           ->  Aggregate  (cost=3.57..3.58 rows=1 width=8) (actual time=0.260..0.271 rows=1 loops=15)
                 ->  HashAggregate  (cost=3.05..3.28 rows=23 width=46) (actual time=0.141..0.192 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=0.681..1.582 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=0.030..0.334 rows=26 loops=1)
                             ->  Hash  (cost=1.23..1.23 rows=23 width=42) (actual time=0.604..0.614 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=0.030..0.312 rows=23 loops=1)
 Planning time: 0.874 ms
 Execution time: 7.417 ms
(28 rows)
```

Output:

f_name	l_name	noofbooks
Thomson	Wayne	5
May-N	Leow	4
Peter	Andreae	3

(3 rows)

To understand the whole nested query, I use stepwise to split this nested query.

```
CREATE VIEW clb AS select f_name, l_name, count(*) as noofbooks
 from customer natural join loaned_book
 group by f_name, l_name;
```

```
FAss2L=> SELECT * FROM clb;
```

f_name	l_name	noofbooks
Jerome	Dolman	1
James	Noble	2
Daisy	Zhou	1
Customer	Default	1
Chang	Chui	1
Ewan	Tempero	2
May-N	Leow	4
Shusen	Yi	1
Nigel	Somerfield	1
Linda	McMurray	1
Kirk	Jackson	1
Peter	Andreae	3
Gang	Xu	1
Thomson	Wayne	5
Craig	Anslow	1

(15 rows)

This table indicates customers and the number of books they borrowed. The purpose of this query is to find out less than three people who borrowed more books than he/she. Simply speaking, it asks for the top three customers. The No 3 customer exactly have 2 people who borrowed more books than him/her. Thus, we can sort this view first and choose the top 3 customers.

```
[FAss2L=>  SELECT * FROM clb ORDER BY noofbooks DESC;
     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)
```

I used 'fetch first 3 rows only' statement to delete rest of the table. Thus, it is the desired result we want to get. Meanwhile, it is exactly same with the result we need to achieve.

```
[FAss2L=>  SELECT * FROM clb ORDER BY noofbooks DESC fetch first 3 rows only;
     f_name      |     l_name      | noofbooks
-----------------+-----------------+-----------
 Thomson         | Wayne           |         5
 May-N           | Leow            |         4
 Peter           | Andreae         |         3
(3 rows)
```

```
[FAss2L=> EXPLAIN ANALYZE SELECT * FROM clb ORDER BY noofbooks DESC fetch first 3 rows only;
                                                            QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=3.81..3.82 rows=3 width=40) (actual time=2.396..2.614 rows=3 loops=1)
   ->  Sort  (cost=3.81..3.87 rows=23 width=40) (actual time=2.371..2.415 rows=3 loops=1)
         Sort Key: (count(*)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  HashAggregate  (cost=3.05..3.28 rows=23 width=40) (actual time=1.976..2.173 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=0.706..1.630 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=0.038..0.355 rows=26 loops=1)
                     ->  Hash  (cost=1.23..1.23 rows=23 width=36) (actual time=0.618..0.640 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=0.031..0.328 rows=23 loops=1)
 Planning time: 0.595 ms
 Execution time: 2.928 ms
(14 rows)
```

After Implementing the above changes, we use explain analyze calculate the new cost. The new cost is very small.

86.03-3.82/86.03 = 95.56% improvement.

Explanation: For this question, from my perspective, the cost drop because I only use (count(*)) once when I create the view. The original statement used three times count(*) function, it increased the cost. Thus, once I get rid of it, the efficiency raised significantly.