300471606 SUBANKAMO

Question 1. Relational Algebra [40 marks]
symbols:
>select operation = σ
>natural join (connect using common attribute) = *
>projection operator (select subset of attribute)= π
>does not equal = <>

a) [25 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.
Working:
1. retrieve all products where Category=’meat’: σCategory=’meat’(Products)
2. natural join result of step 1 with Supplied_By: *Supplied_By
3. Natural join result from step 2 to Company: *Company
4. Select only attribute Description and name from result of step 3: π Description,Name

Answer:
π Description,Name((σCategory=’meat’(Products)*Supplied_By)*Company)

2) [5 marks] Retrieve the names of all companies who always supply products of category ‘fruit’.
Working:
1. retrieve all products where Category=’meat’: σCategory=fruit(Products)
2. natural join result of step 1 with Supplied_By: *Supplied_By
3. Natural join result from step 2 to Company: *Company
4. Select only attribute name from result of step 3: π Name
5. From result of 4 remove the name of any company who has supplied any products that is not in the category fruit: - π Name ((σCategory <> ’fruit’ (Products) * Supplied_By)*Company)

Answer:
π Name ((σCategory=’fruit’ (Products) * Supplied_By)*Company)
- π Name ((σCategory <> ’fruit’ (Products) * Supplied_By)*Company)

3) [5 marks] Retrieve the descriptions of all products that are supplied by two or more companies.
Working:
1. Group attributes in Supplied_By by PId, Count CId for each group, Select groups where there is 2 or greater CId in each group: (σCId>=2 (PId g count (CId)(Supplied_By))
2. Natural join results of step 1 with Products: * Products
3. Select only attribute Description from result of step 2: π Description

Answer:
π Description ((σCId>=2 (PId g count (CId)(Supplied_By))) * Products)

4) [5 marks] Retrieve the names of companies who have not supplied any product in 2023.

Working:

1. Select all records in Supplied_By where year='2023': (σyear='2023'(Supplied_By)
2. Natural join results of step 1 to Company: *Company
3. Remove all results from step 2 from Company: Company-
4. Select only attribute Name from result of step 3: π Name

Answer:

π Name (Company-((σyear='2023'(Supplied_By))*Company))

5) [5 marks] Retrieve the description of products that have been supplied by companies in Wellington who always supply products with price lower than $100.00.

Working:

1. Select all companies where location is in wellington: (σLocation='Wellington'(Company))
2. Select all record in supplied_by where price<100: (σPrice<100(Supplied_By)
3. Natural join result from step 1 and 2: ((σLocation='Wellington'(Company))*(σPrice<100(Supplied_By)))
4. Natural join result from step 3 to products: *Products
5. Select only attribute description from step 4: π Description
6. Remove any description from result of step 5 where the company is in wellington but they have supplied a product for $100 or more: - π Description((σLocation='Wellington'(Company))*(σPrice>=100(Supplied_By)))*Products)

Answer:

π Description((σLocation='Wellington'(Company))*(σPrice<100(Supplied_By)))*Products)
- π Description((σLocation='Wellington'(Company))*(σPrice>=100(Supplied_By)))*Products)

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

1) π Name ,Phone (Products * (σAmount>1000 (Supplied_By) * Company))

Plain english:

Retrieve the name and phone numbers of all companies that supplies more than a 1000 products

SQL

SELECT name, phone
FROM Company NATURAL JOIN Products NATURAL JOIN Supplied_By
WHERE Amount>1000

2) π name, Description (σprice<10 (Products * (Supplied_By * Company)))
Retrieve the name and description of

Plain english:

Retrieve the name of the company that supplies a product and the description of a product that is supplied for less than $10

SQL

```
SELECT name, description
FROM Company NATURAL JOIN Products NATURAL JOIN Supplied_By
WHERE Price<10
```

3) π CId (σAmount>1000 (Supplied_By)) ∩ π CId (Supplied_By * (σDescription='Cake' (Products)))

Plain english:

Retrieve the CId of the company that supplies more than a 1000 cakes

SQL

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

300471606 SUBANKAMO

<u>Question 2. Heuristic and Cost-Based Query Optimization [40 marks]</u>

<u>a) [20 marks] Heuristic query optimization</u>

1a)[5 marks] Transfer the following query into Relational Algebra.

SELECT StudentId, Name, Grade

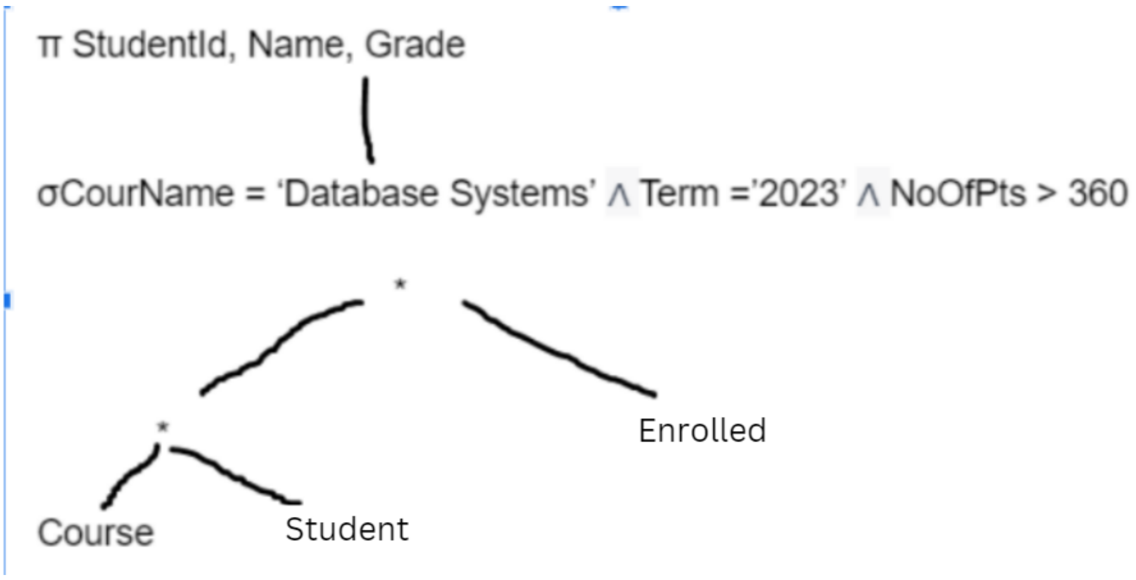
FROM Student NATURAL JOIN Enrolled NATURAL JOIN Course

WHERE CourName = 'Database Systems' AND Term = 2023 AND NoOfPts > 360;

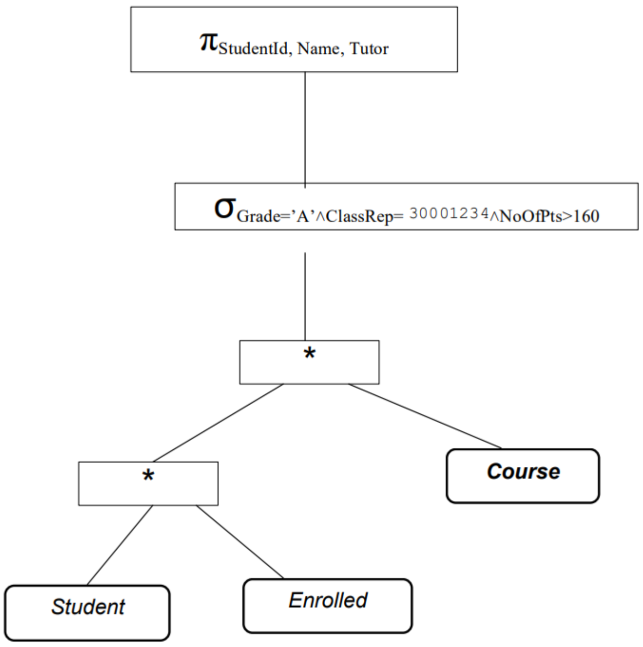
<u>Answer:</u>

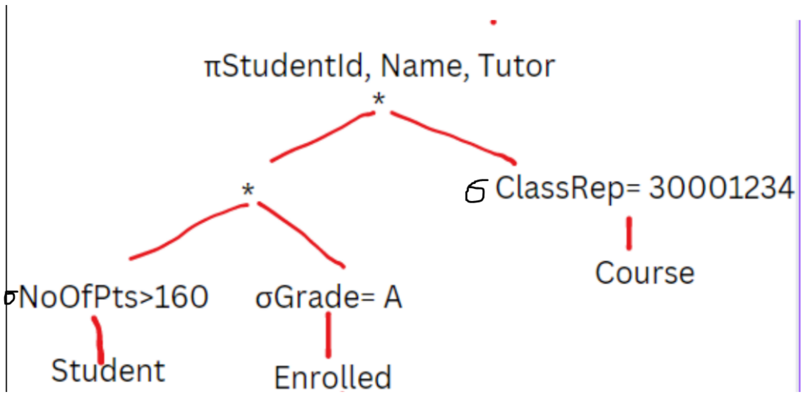
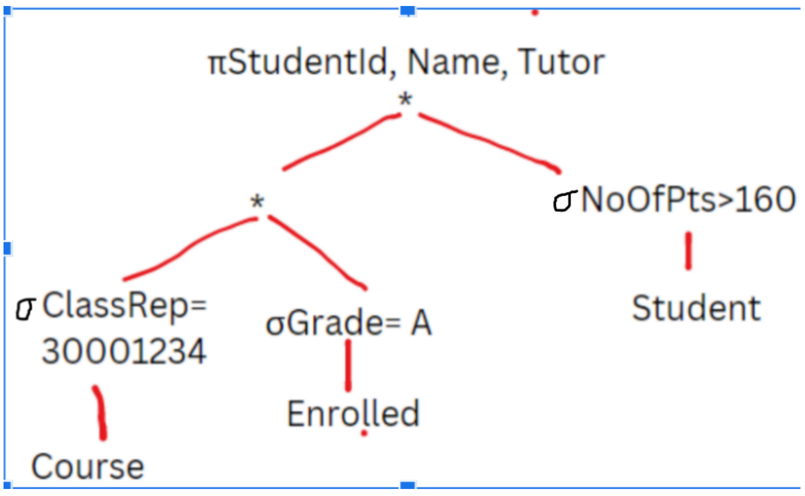
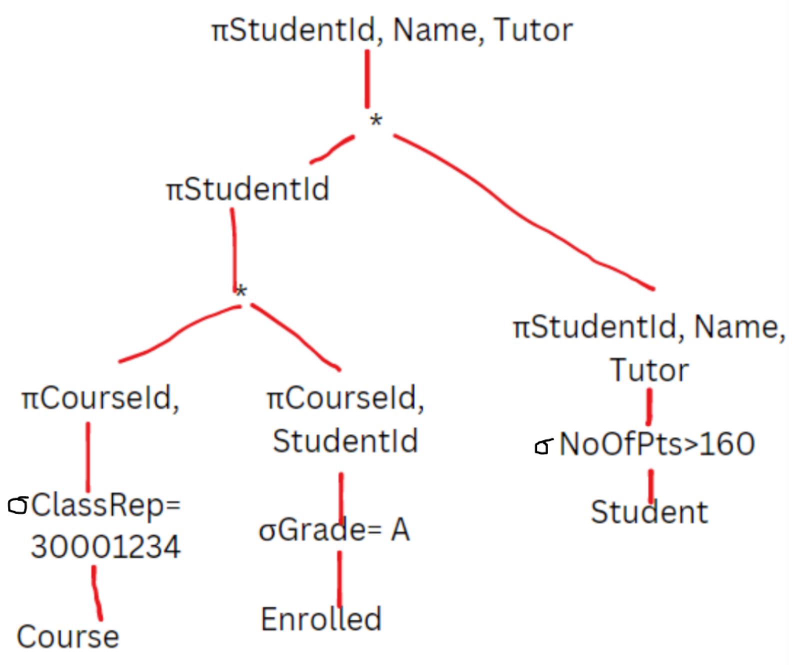
π StudentId, Name, Grade (σCourName = 'Database Systems' ⋀ Term ='2023' ⋀ NoOfPts > 360((Student*Enrolled)*Course))

1b) [3 marks] Draw a query tree for the relational algebra query from 1a


π StudentId, Name, Grade
σCourName = 'Database Systems' ∧ Term ='2023' ∧ NoOfPts > 360
*
*
Enrolled
Course
Student

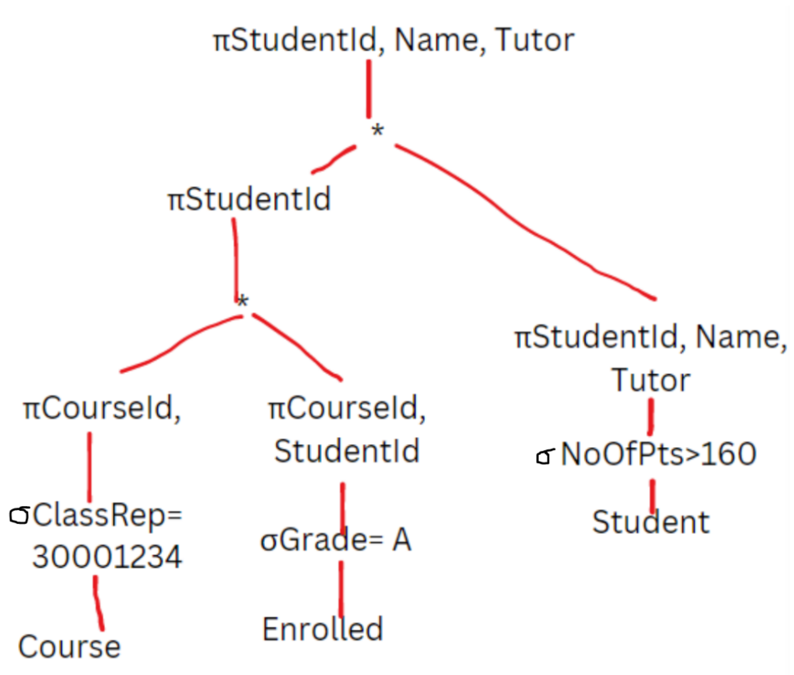
2)[12 marks] Transfer the following query tree into an optimized query tree using the query optimization heuristics.



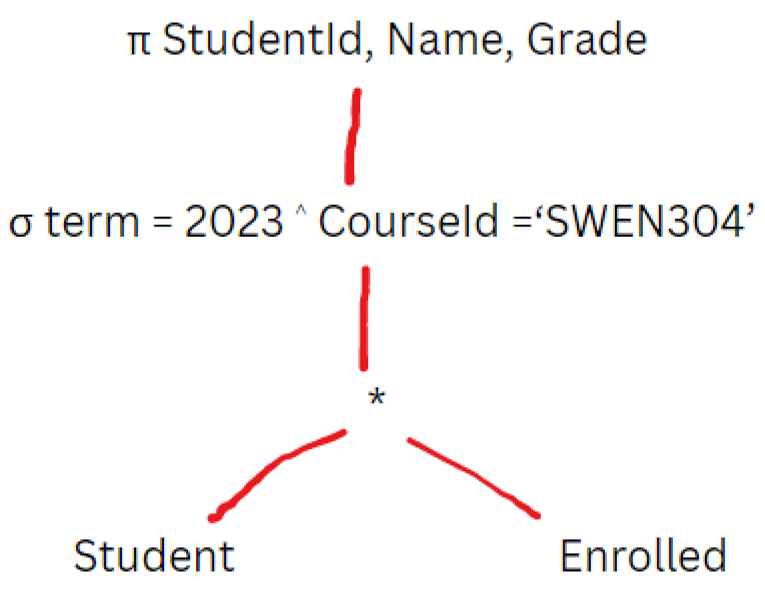
Rule	Description	
6	Move select operations down the tree	 πStudentId, Name, Tutor * * σClassRep= 30001234 Course σNoOfPts>160 σGrade= A Student Enrolled
9	Restrictive select operation applied as early as possible >Assumption: less classRep results in course than NoOfPoint result in Enrolled	 πStudentId, Name, Tutor * * σNoOfPts>160 Student σClassRep= 30001234 σGrade= A Enrolled Course
7	keeping in intermediate relations only the attributes needed by subsequent operations by applying project operations as early as possible	 πStudentId, Name, Tutor * πStudentId * πStudentId, Name, Tutor πCourseId, πCourseId, StudentId σNoOfPts>160 σClassRep= 30001234 Student σGrade= A Course Enrolled

300471606 SUBANKAMO

Answer:



b) [20 marks] Query cost calculation
For each of the given two queries below draw a query tree and calculate the cost of executing query.
(i) π StudentId, Name, Grade (σ term = 2023 ^ CourseId ='SWEN304' (Student * Enrolled))



Student
R Student = (StudentId, Name, NoOfPts, Tutor)

R Student = (int, char(15), smallint, int)
R Student = (4 + 15 + 2 + 4) = 25 Bytes
S Student = R Student * N Student
S Student = 25 * 50000 = 1250000

Enrolled
R Enrolled = (StudentId, CourseId, Term, Grade)
R Enrolled = (int, char(15), smallInt, char(15))
R Enrolled = 4 + 15 + 2 +15 = 36 Bytes
S Enrolled = R Enrolled * N Enrolled
S Enrolled = 36 * 400000 = 14400000

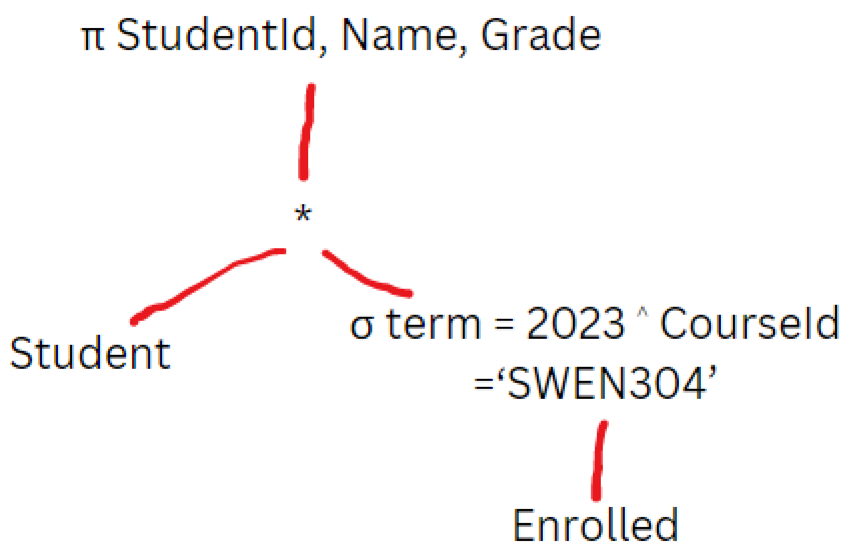
*
R Join = (R Enrolled + R Student - R)
R Join = 36 + 25 - 4 = 57 Bytes
P Join = 1/50000 =0.00002
N Join = N Student * N Enrolled * P Join
N Join =50000*400000*0.00002 = 400000
S Join = N Join * (R Join)
S Join =400000*57 = 22800000

σ term = 2023 ∧ CourseId =‘SWEN304’
P Select =(number of years filtered/number of years)*(number of course filtered/number of courses)
P Select=(1/10)*(1/1000)=0.0001
N Select = N Join * P Select
N Select = 400000 * 0.0001 = 40
S Select = N Select * R Join
S Select = 40 * 57 = 2280

π StudentId, Name, Grade
R Project = (StudentId, Name, Grade)
R Student = (int, char(15), char(15))
R Project = (4 + 15 + 15) = 34
S Project = N Select * R Project
S Project = 40 * 34 = 1360

Answer:
TOTAL COST = 1250000 + 14400000 + 22800000 + 2280 + 1360 = 38,453,640 Bytes

(ii) π StudentId, Name, Grade (Student * σ term = 2023 ∧ CourseId =‘SWEN304’ (Enrolled))



<u>Enrolled</u>
R Enrolled = (StudentId, CourseId, Term, Grade)
R Enrolled = (int, char(15), smallInt, char(15))
R Enrolled = 4 + 15 + 2 +15 = 36 Bytes
S Enrolled = R Enrolled * N Enrolled
S Enrolled = 36 * 400000 = 14400000

<u>σ term = 2023 ∧ CourseId ='SWEN304'</u>
P Select =(number of years filtered/number of years)*(number of course filtered/number of courses)
P Select=(1/10)*(1/1000)=0.0001
N Select = N Enrolled * P Select
N Select = 400000 * 0.0001 = 40
S Select = N Select * R Enrolled
S Select = 40 * 36 = 1440

<u>Student</u>
R Student = (StudentId, Name, NoOfPts, Tutor)
R Student = (int, char(15), smallint, int)
R Student = (4 + 15 + 2 + 4) = 25 Bytes
S Student = R Student * N Student
S Student = 25 * 50000 = 1250000

<u>*</u>
R Join = (R Enrolled + R Student - R)
R Join = 36 + 25 - 4
R Join = 57 Bytes
P Join = 1/50000 = 0.00002
N Join = N Student * N Enrolled * P Join
N Join =50000*40*0.00002 = 40

S Join = N Join * R Join
S Join =40*57 = 2280

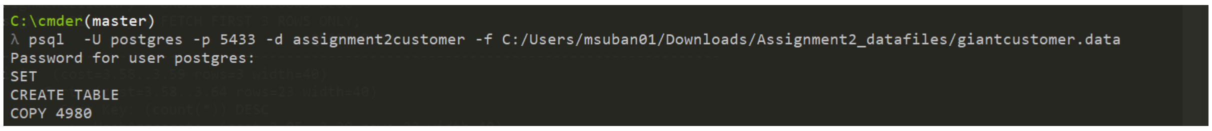
<u>π StudentId, Name, Grade</u>
R Project = (StudentId, Name, Grade)
R Student = (int, char(15), char(15))
R Project = (4 + 15 + 15) = 34
S Project = N Join * R Project
S Project = 40 * 34 = 1360

<u>Answer:</u>
TOTAL COST = 14400000 + 1440 + 1250000 + 2280 + 1360 = 15,655,080 Bytes

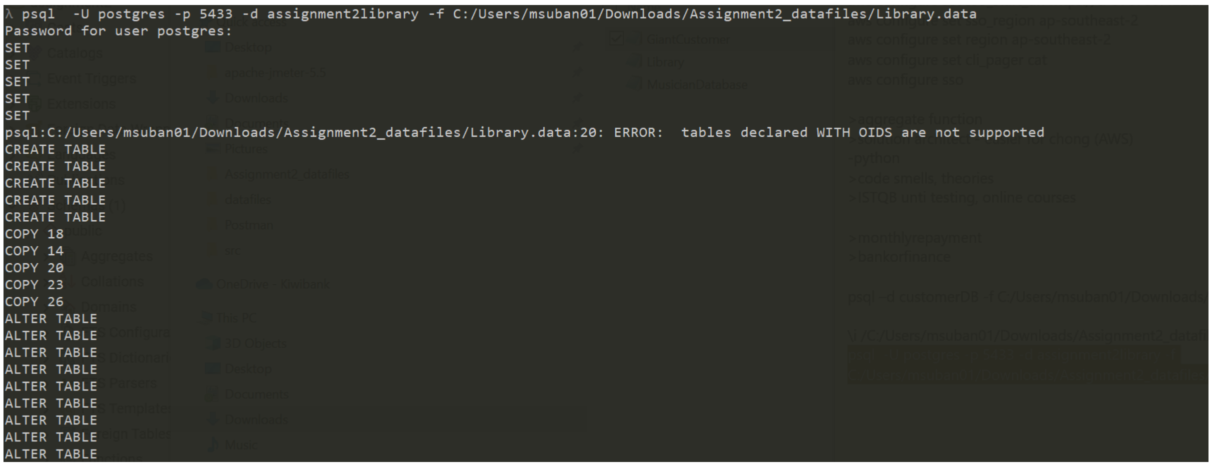
iii) Which of the above two trees has a smaller query cost and why?
The second tree (π StudentId, Name, Grade (Student * σ term = 2023 ^ CourseId ='SWEN304' (Enrolled))) cost 15,655,080 bytes which is 22,798,560 bytes smaller than the first tree (π StudentId, Name, Grade (σ term = 2023 ^ CourseId ='SWEN304' (Student * Enrolled))) which cost 38,453,640 bytes.
The second tree is smaller as it performs select on the enrolled table prior to natural joining the student and enrolled table which means there are less records to join given enrol now only contains records where term = 2023 and CourseId ='SWEN304' .
This results in 40 records being held in the table containing the filtered student and enrolled joined record table instead of 400,000 records.
This aligns with rule 6 of the optimization heuristics where performing select as early as possible (further down the tree) reduces cost.

Question 3. PostgreSQL and Query Optimization [20 marks]

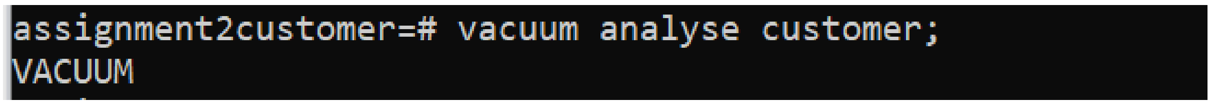
SETUP:



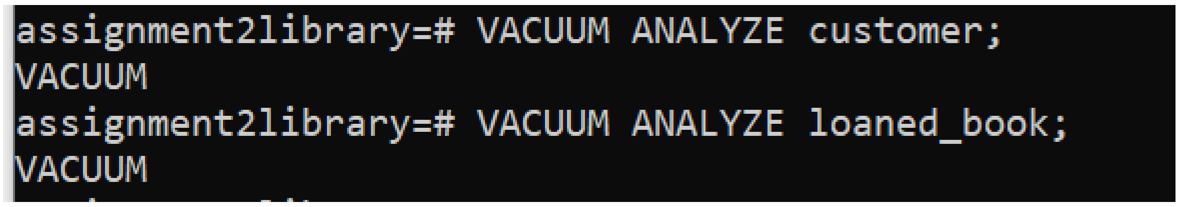
```
C:\cmder(master)
λ psql  -U postgres -p 5433 -d assignment2customer -f C:/Users/msuban01/Downloads/Assignment2_datafiles/giantcustomer.data
Password for user postgres:
SET
CREATE TABLE
COPY 4980
```



```
λ psql  -U postgres -p 5433 -d assignment2library -f C:/Users/msuban01/Downloads/Assignment2_datafiles/Library.data
Password for user postgres:
SET
SET
SET
SET
SET
psql:C:/Users/msuban01/Downloads/Assignment2_datafiles/Library.data:20: ERROR:  tables declared WITH OIDS are not supported
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
COPY 18
COPY 14
COPY 20
COPY 23
COPY 26
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
```



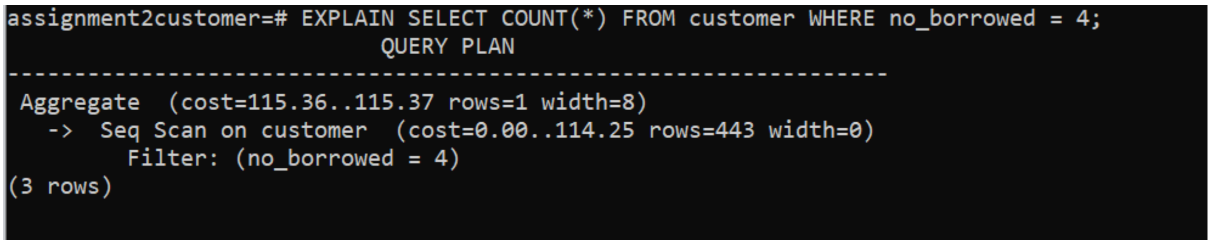
```
assignment2customer=# vacuum analyse customer;
VACUUM
```



```
assignment2library=# VACUUM ANALYZE customer;
VACUUM
assignment2library=# VACUUM ANALYZE loaned_book;
VACUUM
```

a) [6 marks] Improve the cost estimate of the following query:
select count(*) from customer where no_borrowed = 4;

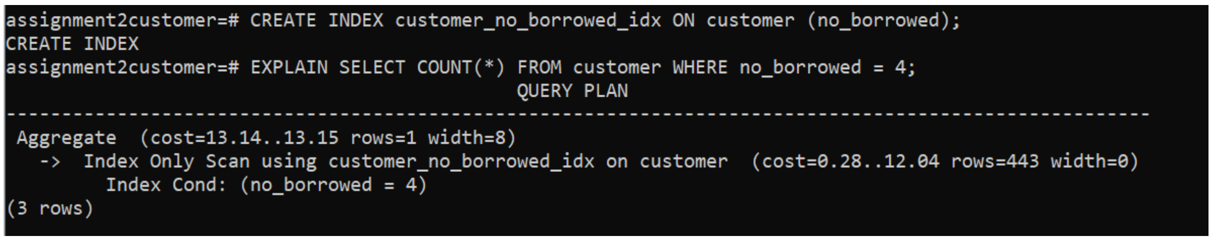
Original cost:
115.37



```
assignment2customer=# EXPLAIN SELECT COUNT(*) FROM customer WHERE no_borrowed = 4;
                               QUERY PLAN
------------------------------------------------------------------
 Aggregate  (cost=115.36..115.37 rows=1 width=8)
   ->  Seq Scan on customer  (cost=0.00..114.25 rows=443 width=0)
         Filter: (no_borrowed = 4)
(3 rows)
```

To improve the cost I will create an index (customer_no_borrowed _idx) on the no_borrowed column in customer

Improved Cost:
13.15.

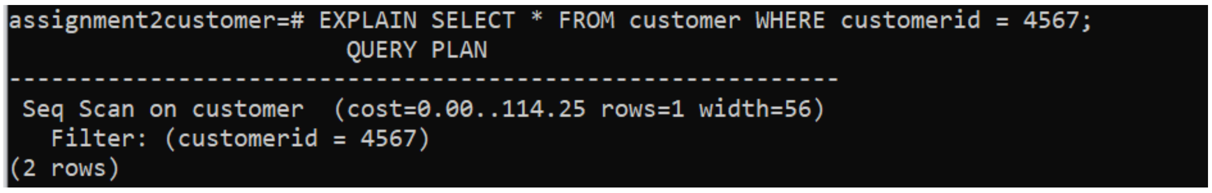


```
assignment2customer=# CREATE INDEX customer_no_borrowed_idx ON customer (no_borrowed);
CREATE INDEX
assignment2customer=# EXPLAIN SELECT COUNT(*) FROM customer WHERE no_borrowed = 4;
                                          QUERY PLAN
------------------------------------------------------------------------------------------------
 Aggregate  (cost=13.14..13.15 rows=1 width=8)
   ->  Index Only Scan using customer_no_borrowed_idx on customer  (cost=0.28..12.04 rows=443 width=0)
         Index Cond: (no_borrowed = 4)
(3 rows)
```

Explanation:
An index is a data structure that allows for faster searching of records in a database. When an index is created on a column, like the no_borrowed column in the customer table, it can reduce the time and cost required to perform searched or sorted operations.

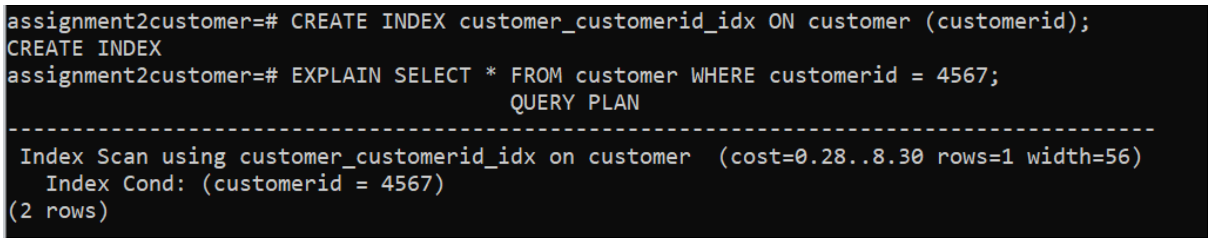
In the original query, PostgreSQL had to scan the entire customer table to find the records where no_borrowed = 4 (sequential search). However, after creating an index on the no_borrowed column, PostgreSQL could use the index to quickly locate the records where no_borrowed = 4 (index search), without having to scan the entire table. This reduces cost by 88.6% ((115.37-13.15)/115.37)

b) [4 marks] Improve the efficiency of the following query: select * from customer where customerid = 4567;
Original Cost:
114.25



```
assignment2customer=# EXPLAIN SELECT * FROM customer WHERE customerid = 4567;
                       QUERY PLAN
-----------------------------------------------------------
 Seq Scan on customer  (cost=0.00..114.25 rows=1 width=56)
   Filter: (customerid = 4567)
(2 rows)
```

To improve the cost I will create an index (customer_customerid_idx) on the customer_id column in customer
Improved Cost:
8.30



```
assignment2customer=# CREATE INDEX customer_customerid_idx ON customer (customerid);
CREATE INDEX
assignment2customer=# EXPLAIN SELECT * FROM customer WHERE customerid = 4567;
                                    QUERY PLAN
----------------------------------------------------------------------------------------
 Index Scan using customer_customerid_idx on customer  (cost=0.28..8.30 rows=1 width=56)
   Index Cond: (customerid = 4567)
(2 rows)
```

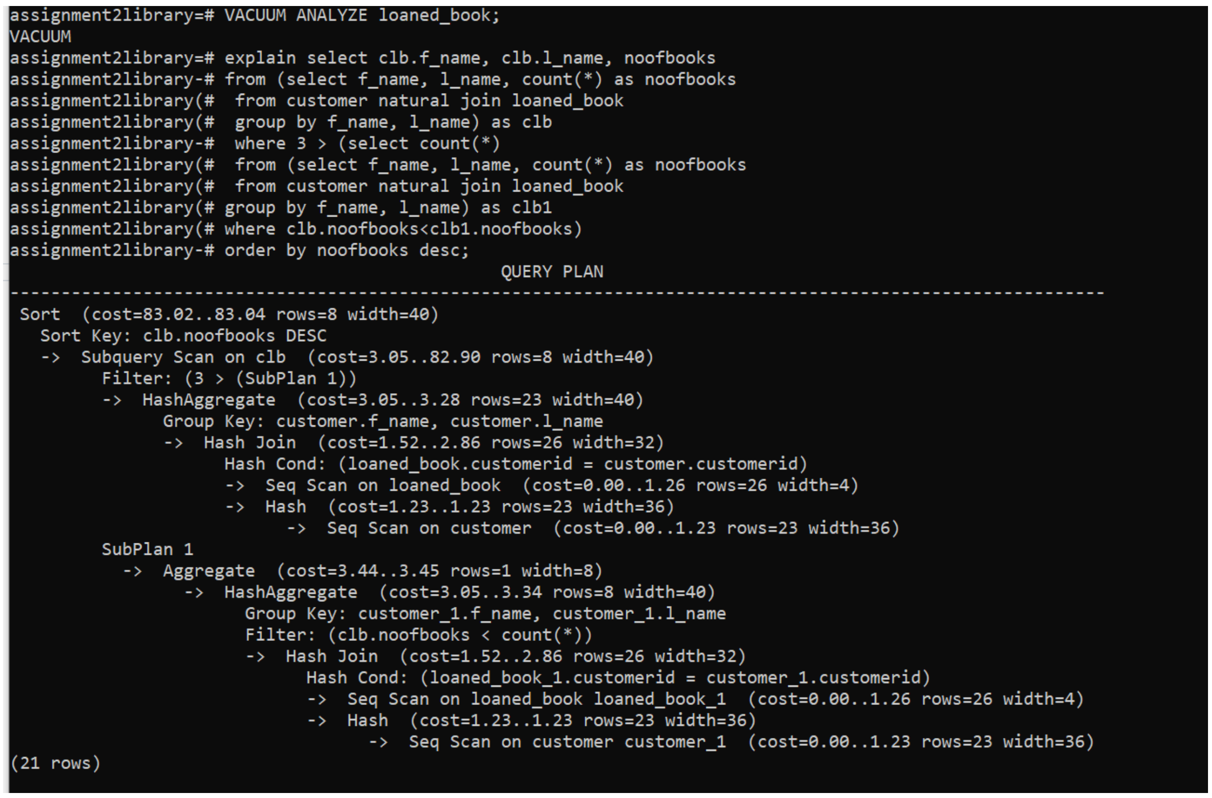
Explanation:
An index is a data structure that allows for faster searching of records in a database. When an index is created on a column, like the customerid column in the customer table, it can reduce the time and cost required to perform searched or sorted operations.

In the original query, PostgreSQL had to scan the entire customer table to find the records where customerid = 4567 (sequential search). However, after creating an index on the customerid column, PostgreSQL could use the index to quickly locate the records where customerid = 4567 (index search), without having to scan the entire table. This reduces cost by 92.7% ((114.25-8.30)/114.25)

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:

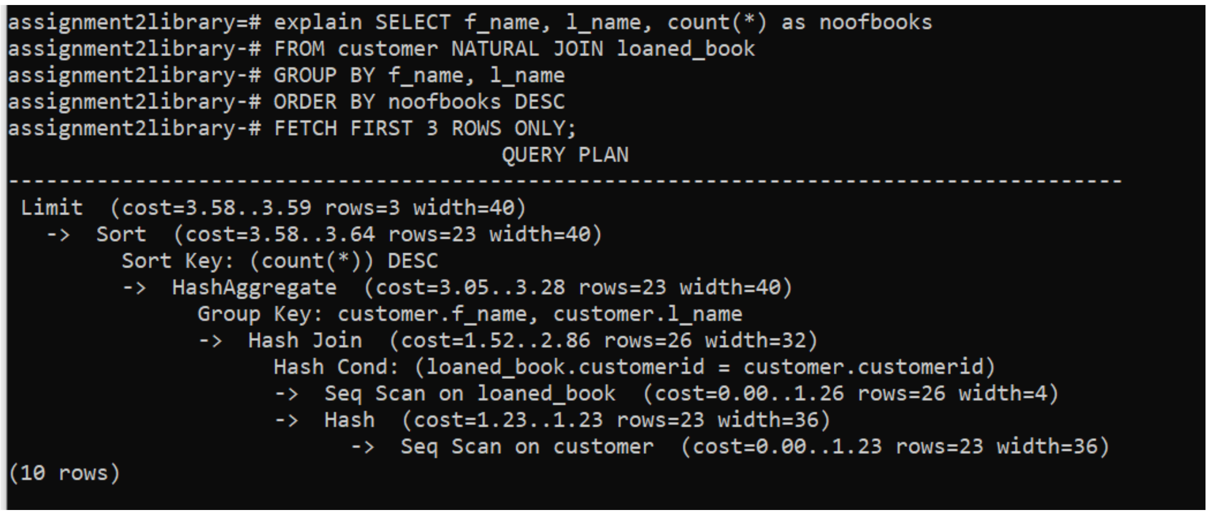
```
select clb.f_name, clb.l_name, noofbooks
from (select f_name, l_name, count(*) as noofbooks
          from customer natural join loaned_book
          group by f_name, l_name) as clb
          where 3 > (select count(*)
                    from (select f_name, l_name, count(*) as noofbooks
     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

Original Cost:
83.04

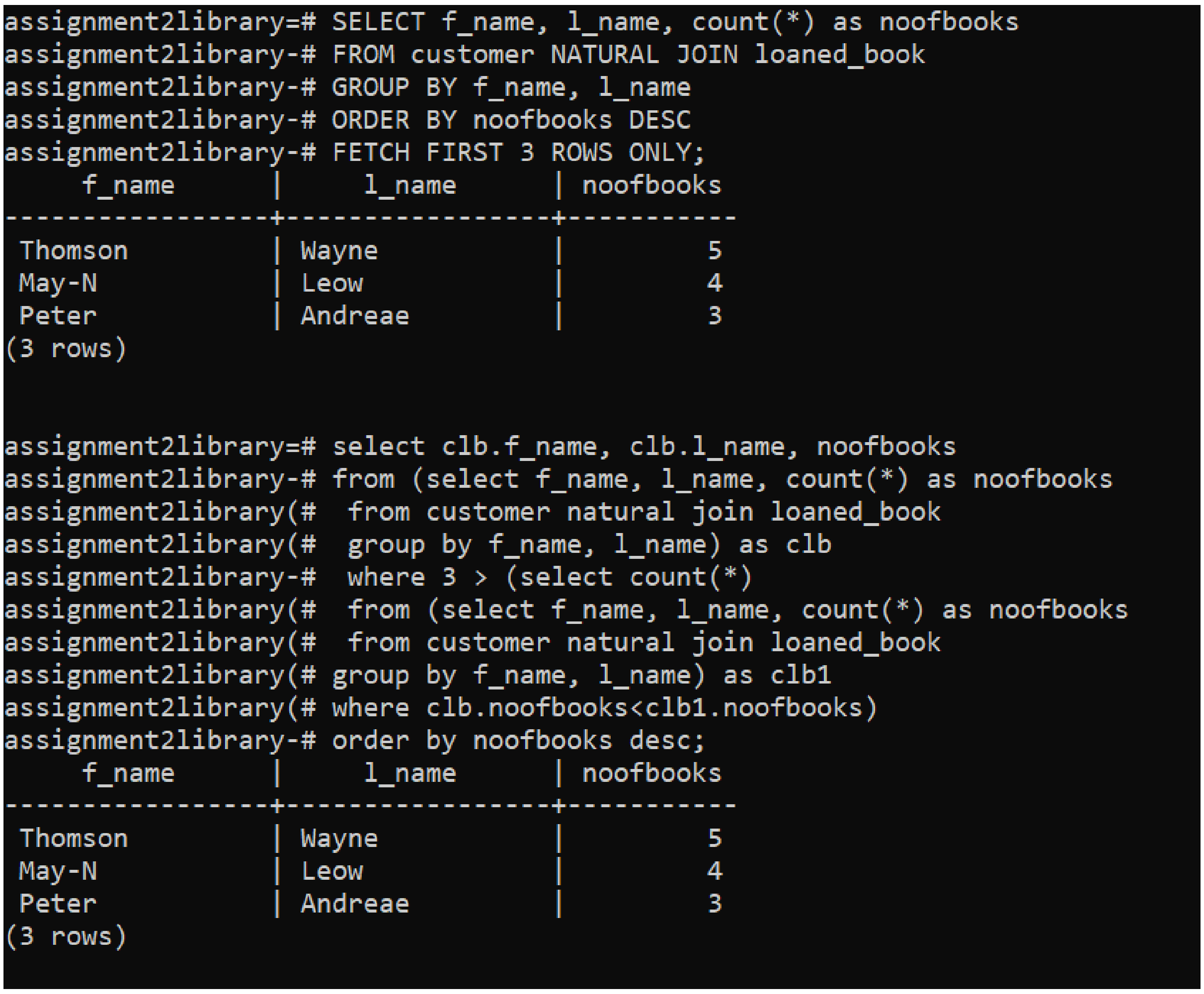


```
assignment2library=# VACUUM ANALYZE loaned_book;
VACUUM
assignment2library=# explain select clb.f_name, clb.l_name, noofbooks
assignment2library-# from (select f_name, l_name, count(*) as noofbooks
assignment2library(#  from customer natural join loaned_book
assignment2library(#  group by f_name, l_name) as clb
assignment2library-#  where 3 > (select count(*)
assignment2library(#  from (select f_name, l_name, count(*) as noofbooks
assignment2library(#  from customer natural join loaned_book
assignment2library(# group by f_name, l_name) as clb1
assignment2library(# where clb.noofbooks<clb1.noofbooks)
assignment2library-# order by noofbooks desc;
                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Sort  (cost=83.02..83.04 rows=8 width=40)
   Sort Key: clb.noofbooks DESC
   ->  Subquery Scan on clb  (cost=3.05..82.90 rows=8 width=40)
         Filter: (3 > (SubPlan 1))
         ->  HashAggregate  (cost=3.05..3.28 rows=23 width=40)
               Group Key: customer.f_name, customer.l_name
               ->  Hash Join  (cost=1.52..2.86 rows=26 width=32)
                     Hash Cond: (loaned_book.customerid = customer.customerid)
                     ->  Seq Scan on loaned_book  (cost=0.00..1.26 rows=26 width=4)
                     ->  Hash  (cost=1.23..1.23 rows=23 width=36)
                           ->  Seq Scan on customer  (cost=0.00..1.23 rows=23 width=36)
         SubPlan 1
           ->  Aggregate  (cost=3.44..3.45 rows=1 width=8)
                 ->  HashAggregate  (cost=3.05..3.34 rows=8 width=40)
                       Group Key: customer_1.f_name, customer_1.l_name
                       Filter: (clb.noofbooks < count(*))
                       ->  Hash Join  (cost=1.52..2.86 rows=26 width=32)
                             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)
                             ->  Hash  (cost=1.23..1.23 rows=23 width=36)
                                   ->  Seq Scan on customer customer_1  (cost=0.00..1.23 rows=23 width=36)
(21 rows)
```

Improved Cost:
3.59



```
assignment2library=# explain SELECT f_name, l_name, count(*) as noofbooks
assignment2library-# FROM customer NATURAL JOIN loaned_book
assignment2library-# GROUP BY f_name, l_name
assignment2library-# ORDER BY noofbooks DESC
assignment2library-# FETCH FIRST 3 ROWS ONLY;
                                       QUERY PLAN
---------------------------------------------------------------------------------------
 Limit  (cost=3.58..3.59 rows=3 width=40)
   ->  Sort  (cost=3.58..3.64 rows=23 width=40)
         Sort Key: (count(*)) DESC
         ->  HashAggregate  (cost=3.05..3.28 rows=23 width=40)
               Group Key: customer.f_name, customer.l_name
               ->  Hash Join  (cost=1.52..2.86 rows=26 width=32)
                     Hash Cond: (loaned_book.customerid = customer.customerid)
                     ->  Seq Scan on loaned_book  (cost=0.00..1.26 rows=26 width=4)
                     ->  Hash  (cost=1.23..1.23 rows=23 width=36)
                           ->  Seq Scan on customer  (cost=0.00..1.23 rows=23 width=36)
(10 rows)
```



```
assignment2library=# SELECT f_name, l_name, count(*) as noofbooks
assignment2library-# FROM customer NATURAL JOIN loaned_book
assignment2library-# GROUP BY f_name, l_name
assignment2library-# ORDER BY noofbooks DESC
assignment2library-# FETCH FIRST 3 ROWS ONLY;
     f_name      |     l_name      | noofbooks
-----------------+-----------------+-----------
 Thomson         | Wayne           |         5
 May-N           | Leow            |         4
 Peter           | Andreae         |         3
(3 rows)


assignment2library=# select clb.f_name, clb.l_name, noofbooks
assignment2library-# from (select f_name, l_name, count(*) as noofbooks
assignment2library(#  from customer natural join loaned_book
assignment2library(#  group by f_name, l_name) as clb
assignment2library-#  where 3 > (select count(*)
assignment2library(#  from (select f_name, l_name, count(*) as noofbooks
assignment2library(#  from customer natural join loaned_book
assignment2library(# group by f_name, l_name) as clb1
assignment2library(# where clb.noofbooks<clb1.noofbooks)
assignment2library-# order by noofbooks desc;
     f_name      |     l_name      | noofbooks
-----------------+-----------------+-----------
 Thomson         | Wayne           |         5
 May-N           | Leow            |         4
 Peter           | Andreae         |         3
(3 rows)
```

Explanation:

The given query is inefficient because it involves multiple subqueries and joins, which can result in a large number of operations and high resource usage.
In the original query there are two nested subqueries, where the inner subquery is identical to the first query above, and the outer subquery filters the results to return only customers who have borrowed fewer books than at least three other customers. This requires comparing the count of books borrowed by each customer with the count of books borrowed by all other customers, which can be a very expensive operation for large tables.

One way to improve the efficiency of this query is to only have one natural join between customer and loaned_book table, group records that have the same f_name and l_name attribute value, order by the number of records in each group and return the first three result (top 3 will have less than 3 customers borrowing more books lower than top 3 will have 3 or more). This approach avoids the need for multiple subqueries and joins, and can therefore be more efficient.
This reduces cost by 95.7% ((83.04-3.59)/83.04)
