P3answers

Exercise 1

a)

Command:

\d+ customers

Answer:

There is 1 index called “customers_pkey”

b)

Command:

\d+

\di+

Answer:

Two relations occupy the most number of disk pages:

customers

orderlines

Two indexes that occupy the most number of disk pages:

customers_pkey

orders_pkey

c)

Command:

SELECT count(*) FROM customers;

SELECT attname, n_distinct FROM pg_stats WHERE tablename = 'customers';

Answer:

```
 count
-------
 20000
(1 row)
```

```
     attname      | n_distinct
------------------+------------
 customerid       |         -1
 firstname        |         -1
 lastname         |         -1
 address1         |         -1
 address2         |          1
 city             |         -1
 state            |         52
 zip              |     -0.475
 country          |         11
 region           |          2
 email            |         -1
 phone            |         -1
 creditcardtype   |          5
```

```
 creditcard                |              -1
 creditcardexpiration |              60
 username                  |              -1
 password                  |               1
 age                       |              73
 income                    |               5
 gender                    |               2
(20 rows)
```

For tuples with negative n_distinct, we first negate the number and then multiply it with the total entry of the customers table, which is 20000

So the result is

```
          attname          | n_distinct
----------------------+------------
 customerid                |      20000
 firstname                 |      20000
 lastname                  |      20000
 address1                  |      20000
 address2                  |          1
 city                      |      20000
 state                     |         52
 zip                       |       9500
 country                   |         11
 region                    |          2
 email                     |      20000
 phone                     |      20000
 creditcardtype            |          5
 creditcard                |      20000
 creditcardexpiration |         60
 username                  |      20000
 password                  |          1
 age                       |         73
 income                    |          5
 gender                    |          2
```

Except for the customerid attibute, firstname or phone number are suitable for B-tree indexes.

d)

Command:

```
SELECT COUNT (DISTINCT customerid) AS customerid,
        COUNT (DISTINCT firstname) AS firstname,
        COUNT (DISTINCT lastname) AS lastname,
        COUNT (DISTINCT address1) AS address1,
        COUNT (DISTINCT address2) AS address2,
        COUNT (DISTINCT city) AS city,
        COUNT (DISTINCT state) AS state,
        COUNT (DISTINCT zip) AS zip,
        COUNT (DISTINCT country) AS country,
```

```
        COUNT (DISTINCT region) AS region,
        COUNT (DISTINCT email) AS email,
        COUNT (DISTINCT phone) AS phone,
        COUNT (DISTINCT creditcardtype) AS creditcardtype,
        COUNT (DISTINCT creditcard) AS creditcard,
        COUNT (DISTINCT creditcardexpiration) AS creditcardexpiration,
        COUNT (DISTINCT username) AS username,
        COUNT (DISTINCT password) AS password,
        COUNT (DISTINCT age) AS age,
        COUNT (DISTINCT income) AS income,
        COUNT (DISTINCT gender) AS gender
FROM customers;
```

Answer:

```
 customerid | firstname | lastname | address1 | address2 | city    | state | zip    | country | region | email | phone | creditcardtype
| creditcard | creditcardexpiration | username | password | age | income | gender
------------+-----------+----------+----------+----------+-------+-------+------+---------+--------+-------+-------+----------------+-------
-----+----------------------+----------+----------+-----+--------+--------
         20000 |        20000 |       20000 |       20000 |             1 | 20000 |       52 | 9500 |          11 |          2 | 20000 |
20000 |                    5 |         20000 |                         60 |       20000 |             1 |    73 |          5 |          2
(1 row)
```

The answer is exactly the same and the comparison table is as follow:

attname	from catalog	from sql query
customerid	20000	20000
firstname	20000	20000
lastname	20000	20000
address1	20000	20000
address2	1	1
city	20000	20000
state	52	52
zip	9500	9500
country	11	11
region	2	2
email	20000	20000
phone	20000	20000
creditcardtype	5	5
creditcard	20000	20000
creditcardexpiration	60	60
username	20000	20000
password	1	1
age	73	73
income	5	5
gender	2	2

Exercise 2

a)

Command:

```
EXPLAIN SELECT * FROM customers WHERE country = 'Japan';
SELECT COUNT(*) FROM customers WHERE country = 'Japan';
```

Answer:

```
                                QUERY PLAN
-----------------------------------------------------------------
  Seq Scan on customers    (cost=0.00..721.00 rows=995 width=156)
     Filter: ((country)::text = 'Japan'::text)
(2 rows)
```

Use SELECT COUNT(*) FROM customers WHERE country = 'Japan' query

```
  count
-------
     995
(1 row)
```

The estimated value is the same as the actual value

b)

Command:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'customers';
```

Answer:

Use the formula (disk_pages_read * seg_page_cost) + (rows scanned * cpu_tuple_cost) and the default value for seg_page_cost is 1.0 and for cpu_tuple_cost is 0.01

Since there is a filter by WHERE clause, we have to add rows scanned * cpu_operator_cost, of which the default value is 0.0025.

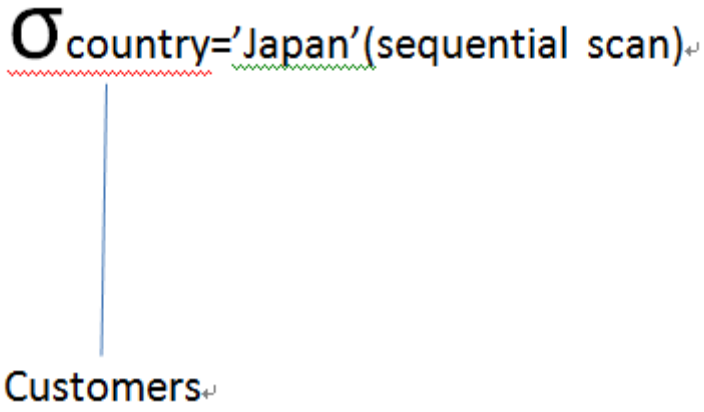
Use SELECT relpages, reltuples FROM pg_class WHERE relname = 'customers';

The answer is as follow

```
  relpages | reltuples
----------+-----------
         471 |         20000
(1 row)
```

So the formula is (471 * 1.0) + (20000 * 0.01) + (20000 * 0.0025) = 721

c)



Excercise 3

a)

Command:

SELECT relpages FROM pg_class WHERE relname = 'customers_country';

Answer:

```
 relpages
----------
        59
(1 row)
```

So it takes 59 pages

b)

Command:

EXPLAIN SELECT * FROM customers WHERE country = 'Japan';

Answer:

Rerun EXPLAIN from Exercise 2, we find the answer is the sanme, meaning that the query optimizer still chooses sequential scan as the best plan and the total cost is 721.00

c)

Command:

CLUSTER customers_country ON customers;
VACUUM;
ANALYZE;

Answer:

It is better not to use unclustered index for the query.
The possible reasons are as follow:
First, since we add index and it takes 59 pages so when we want to get the qualifying entry, there is possiblility to check from one page to another, which will take extra time
Second, because it is unclustered, the qualifying data is not consecutive and notice that there is a lot of data distributing in many pages so we still have to check every page so the unclustered index has no help on saving cost

d)

Command:

EXPLAIN SELECT * FROM customers WHERE country = 'Japan';

Answer:

```
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using customers_country on customers  (cost=0.00..56.66 rows=995 width=156)
   Index Cond: ((country)::text = 'Japan'::text)
(2 rows)
```

e)

Answer:

If we change unclustered index into clustered index, the best plan is to use clustered index but with adding unclustered index, the best plan is sequential scan.

The difference is that for clustered index, the order of data records is the same as or close to the order of data entries. Therefore, the data qualifying the condition of country = 'Japan' would gather in the same or closed page, which will decrease the page I/O.

Exercise 4

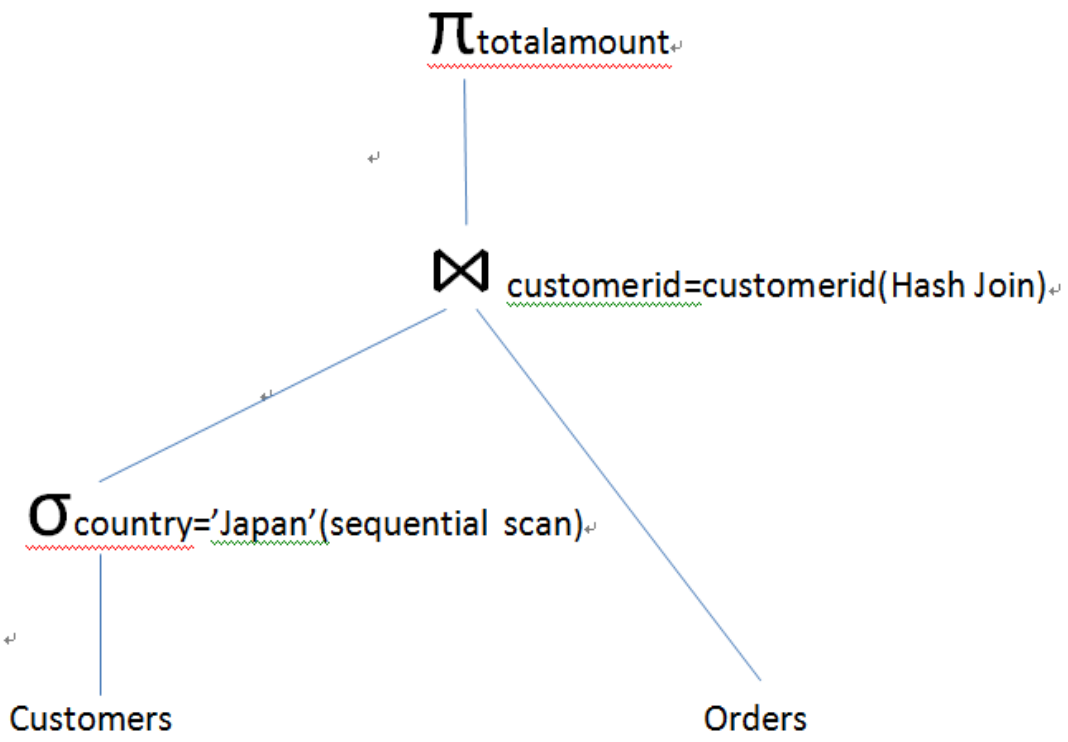
a)

Command:

```
EXPLAIN SELECT totalamount
      FROM Customers C, Orders O
      WHERE C.customerid = O.customerid AND C.country = 'Japan';
```

Answer:

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=733.44..1004.41 rows=597 width=8)
   Hash Cond: (o.customerid = c.customerid)
   ->  Seq Scan on orders o  (cost=0.00..220.00 rows=12000 width=12)
   ->  Hash  (cost=721.00..721.00 rows=995 width=4)
         ->  Seq Scan on customers c  (cost=0.00..721.00 rows=995 width=4)
               Filter: ((country)::text = 'Japan'::text)
(6 rows)
```



b)

Answer:

It uses hash join.

The total estimated cost for the join process is 1004.41

The estimated result cardinality is 597

c)

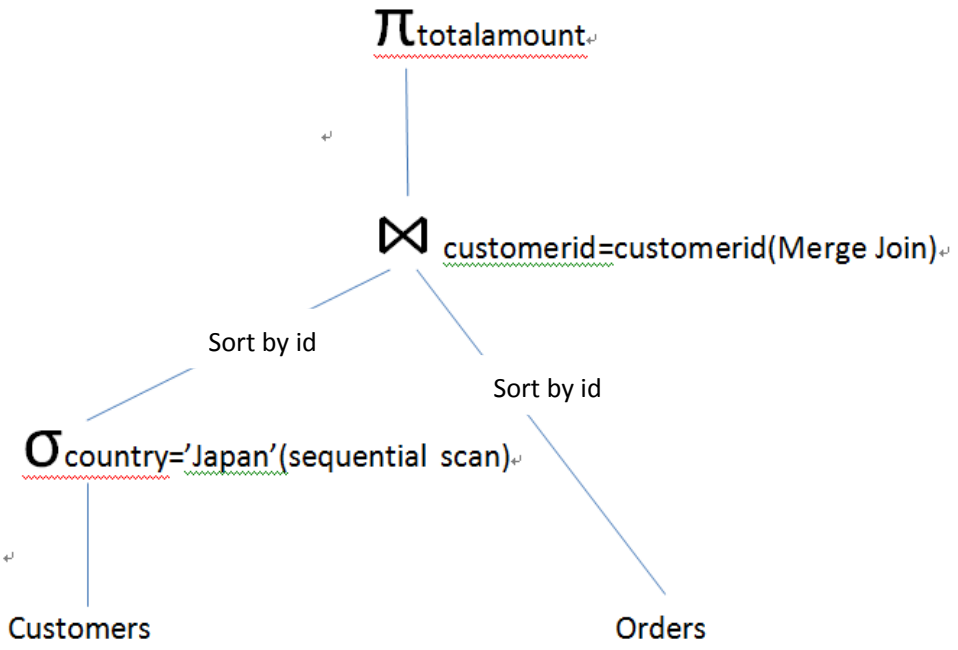
Command:

```
SET enable_hashjoin TO off;
EXPLAIN SELECT totalamount
      FROM Customers C, Orders O
      WHERE C.customerid = O.customerid AND C.country = 'Japan';
```

Answer:

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Merge Join    (cost=1803.59..1874.53 rows=597 width=8)
   Merge Cond: (c.customerid = o.customerid)
   ->  Sort    (cost=770.54..773.03 rows=995 width=4)
         Sort Key: c.customerid
         ->  Seq Scan on customers c    (cost=0.00..721.00 rows=995 width=4)
               Filter: ((country)::text = 'Japan'::text)
   ->  Sort    (cost=1033.04..1063.04 rows=12000 width=12)
         Sort Key: o.customerid
         ->  Seq Scan on orders o    (cost=0.00..220.00 rows=12000 width=12)
(9 rows)
```

It uses Merge join.
The total cost is 1874.53



d)

Command:

```
SET enable_mergejoin TO off;
EXPLAIN SELECT totalamount
      FROM Customers C, Orders O
      WHERE C.customerid = O.customerid AND C.country = 'Japan';
```

Answer:

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
```

```
 Nested Loop  (cost=0.00..5749.36 rows=597 width=8)
   ->  Seq Scan on orders o  (cost=0.00..220.00 rows=12000 width=12)
   ->  Index Scan using customers_pkey on customers c  (cost=0.00..0.45 rows=1 width=4)
         Index Cond: (c.customerid = o.customerid)
         Filter: ((c.country)::text = 'Japan'::text)
(5 rows)
```

It uses Nested Loop
The total cost is 5749.36

Exercise 5

a)

Command:

```
\i setup_db.sql
VACUUM;
ANALYZE;

SET enable_hashjoin TO on;
SET enable_mergejoin TO on;

EXPLAIN SELECT AVG(totalamount) as avgOrder, country
      FROM Customers C, Orders O
      WHERE C.customerid = O.customerid
      GROUP BY country
      ORDER BY avgOrder;

SET enable_hashjoin TO off;

EXPLAIN SELECT AVG(totalamount) as avgOrder, country
      FROM Customers C, Orders O
      WHERE C.customerid = O.customerid
      GROUP BY country
      ORDER BY avgOrder;
```

Answer:

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------
 Sort  (cost=1501.33..1501.36 rows=11 width=13)
   Sort Key: (avg(o.totalamount))
   ->  HashAggregate  (cost=1501.00..1501.14 rows=11 width=13)
         ->  Hash Join  (cost=921.00..1441.00 rows=12000 width=13)
               Hash Cond: (o.customerid = c.customerid)
               ->  Seq Scan on orders o  (cost=0.00..220.00 rows=12000 width=12)
               ->  Hash  (cost=671.00..671.00 rows=20000 width=9)
                     ->  Seq Scan on customers c  (cost=0.00..671.00 rows=20000 width=9)
(8 rows)
```

It uses Hash join.
The total cost is 1501.36

```
                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------
----
 Sort  (cost=2325.52..2325.55 rows=11 width=13)
   Sort Key: (avg(o.totalamount))
   ->  HashAggregate  (cost=2325.19..2325.33 rows=11 width=13)
         ->  Merge Join  (cost=1033.15..2265.19 rows=12000 width=13)
               Merge Cond: (c.customerid = o.customerid)
               ->  Index Scan using customers_pkey on customers c  (cost=0.00..1002.25 rows=20000 width
=9)
               ->  Sort  (cost=1033.04..1063.04 rows=12000 width=12)
                     Sort Key: o.customerid
                     ->  Seq Scan on orders o  (cost=0.00..220.00 rows=12000 width=12)
(9 rows)
```

It uses Merge join.
The total cost is 2325.55

b)
Command:

```
SET enable_hashjoin TO on;

EXPLAIN SELECT *
      FROM Customers C, Orders O
      WHERE C.customerid = O.customerid
      ORDER BY C.customerid;

SET enable_mergejoin TO off;

EXPLAIN SELECT *
      FROM Customers C, Orders O
      WHERE C.customerid = O.customerid
      ORDER BY C.customerid;
```

Answer:

```
                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Merge Join  (cost=1033.15..2265.19 rows=12000 width=192)
   Merge Cond: (c.customerid = o.customerid)
   ->  Index Scan using customers_pkey on customers c  (cost=0.00..1002.25 rows=20000 width=156)
   ->  Sort  (cost=1033.04..1063.04 rows=12000 width=36)
         Sort Key: o.customerid
         ->  Seq Scan on orders o  (cost=0.00..220.00 rows=12000 width=36)
(6 rows)
```

It uses Merge join.
The total cost is 2265.19

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Sort  (cost=3783.54..3813.54 rows=12000 width=192)
   Sort Key: c.customerid
   ->  Hash Join  (cost=370.00..1861.00 rows=12000 width=192)
         Hash Cond: (c.customerid = o.customerid)
         ->  Seq Scan on customers c  (cost=0.00..671.00 rows=20000 width=156)
         ->  Hash  (cost=220.00..220.00 rows=12000 width=36)
               ->  Seq Scan on orders o  (cost=0.00..220.00 rows=12000 width=36)
(7 rows)
```

It uses Hash join.
The total cost is 3813.54

c)
Answer:
For 5.1, the optimizer make judgement that merge join takes longer time than hash join, but the queries do not need to take advantage of the fact that entries is sorted by customerid after merge. So hash join will take less time.
For 5.2, we make queries based on the order of customerid and after merge, the customerid is already sorted, the query can now take advantage of this. So the merge join is preferred.

Exercise 6
a)
Command:

```
\i setup_db.sql
VACUUM;
ANALYZE;

EXPLAIN SELECT C.customerid, C.lastname
FROM Customers C
WHERE
4 < (
      SELECT COUNT(*)
      FROM Orders O
      WHERE O.customerid = C.customerid);
```

Answer:

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on customers c  (cost=0.00..5001021.00 rows=6667 width=15)
   Filter: (4 < (SubPlan 1))
   SubPlan 1
     ->  Aggregate  (cost=250.00..250.01 rows=1 width=0)
           ->  Seq Scan on orders o  (cost=0.00..250.00 rows=1 width=0)
                 Filter: (customerid = $0)
(6 rows)
```

The total cost is 5001021.00

b)

Command:

```
CREATE VIEW OrderCount AS
        SELECT O.customerid, count(*) AS numorders
        FROM Orders O
        GROUP BY O.customerid;
```

c)

Command:

```
SELECT C.customerid, C.lastname
FROM Customers C, OrderCount O
WHERE C.customerid = O.customerid AND O.numorders > 4;
```

d)

Command:

```
EXPLAIN SELECT C.customerid, C.lastname
    FROM Customers C, OrderCount O
    WHERE C.customerid = O.customerid AND O.numorders > 4;
```

Answer:

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=1231.00..1703.29 rows=8996 width=15)
   Hash Cond: (o.customerid = c.customerid)
   ->  HashAggregate  (cost=310.00..467.43 rows=8996 width=4)
         Filter: (count(*) > 4)
         ->  Seq Scan on orders o  (cost=0.00..220.00 rows=12000 width=4)
   ->  Hash  (cost=671.00..671.00 rows=20000 width=15)
         ->  Seq Scan on customers c  (cost=0.00..671.00 rows=20000 width=15)
(7 rows)
```

The total cost is 1703.29

It is much smaller than the cost from part (a)

Exercise 7

a)

Command:

```
\i setup_db.sql
VACUUM;
ANALYZE;

EXPLAIN SELECT customerid, lastname, numorders
    FROM
    (
        SELECT C.customerid, C.lastname, count(*) as numorders
        FROM Customers C, Orders O
        WHERE C.customerid = O.customerid AND C.country = 'Japan'
        GROUP BY C.customerid, lastname
```

```
) AS ORDERCOUNTS1
WHERE 5 >=
(
    SELECT count(*)
    FROM
    (
        SELECT C.customerid, C.lastname, count(*) as numorders
        FROM Customers C, Orders O
        WHERE C.customerid=O.customerid AND C.country = 'Japan'
        GROUP BY C.customerid, lastname
    ) AS ORDERCOUNTS2
    WHERE ORDERCOUNTS1.numorders < ORDERCOUNTS2.numorders
)
ORDER BY customerid;
```

Answer:

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Sort  (cost=614926.51..614927.01 rows=199 width=130)
   Sort Key: ordercounts1.customerid
   ->  Subquery Scan ordercounts1  (cost=1008.88..614918.91 rows=199 width=130)
         Filter: (5 >= (SubPlan 1))
         ->  HashAggregate  (cost=1008.88..1016.35 rows=597 width=15)
               ->  Hash Join  (cost=733.44..1004.41 rows=597 width=15)
                     Hash Cond: (o.customerid = c.customerid)
                     ->  Seq Scan on orders o  (cost=0.00..220.00 rows=12000 width=4)
                     ->  Hash  (cost=721.00..721.00 rows=995 width=15)
                           ->  Seq Scan on customers c  (cost=0.00..721.00 rows=995 width=15)
                                 Filter: ((country)::text = 'Japan'::text)
         SubPlan 1
           ->  Aggregate  (cost=1028.29..1028.30 rows=1 width=0)
                 ->  HashAggregate  (cost=1010.38..1020.83 rows=597 width=15)
                       Filter: ($0 < count(*))
                       ->  Hash Join  (cost=733.44..1004.41 rows=597 width=15)
                             Hash Cond: (o.customerid = c.customerid)
                             ->  Seq Scan on orders o  (cost=0.00..220.00 rows=12000 width=4)
                             ->  Hash  (cost=721.00..721.00 rows=995 width=15)
                                   ->  Seq Scan on customers c  (cost=0.00..721.00 rows=995 width=15)
                                         Filter: ((country)::text = 'Japan'::text)
(21 rows)
```

The total cost is 614927.01

b)

Command:

```
CREATE VIEW OrderCountJapan AS
        SELECT O.customerid, count(*) AS numorders
        FROM Orders O, Customers C
        WHERE C.customerid = O.customerid AND C.country='Japan'
```

```
GROUP BY O.customerid;

CREATE VIEW MoreFrequentJapanCustomers AS
    (
        SELECT O3.customerid, 0 as ORank
        FROM OrderCountJapan O3
        WHERE O3.numorders IN (SELECT MAX(numorders) FROM OrderCountJapan)
    )
    UNION
    (
        SELECT O2.customerid, count(*) as ORank
        FROM OrderCountJapan O2, OrderCountJapan O1
        WHERE O2.numorders < O1.numorders
        GROUP BY O2.customerid
    )
    ORDER BY 2;
```

c)

Command:

```
SELECT C.customerid, C.lastname, O.numorders
FROM Customers C, OrderCountJapan O, MoreFrequentJapanCustomers F
WHERE C.customerid = O.customerid AND C.customerid = F.customerid AND F.ORank <= 5
ORDER BY C.customerid;
```

d)

Command:

```
EXPLAIN SELECT C.customerid, C.lastname, O.numorders
FROM Customers C, OrderCountJapan O, MoreFrequentJapanCustomers F
WHERE C.customerid = O.customerid AND C.customerid = F.customerid AND F.ORank <= 5
ORDER BY C.customerid;
```

Answer:

```
                                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=14629.76..14629.77 rows=5 width=23)
   Sort Key: c.customerid
   ->  Hash Join  (cost=13766.46..14629.70 rows=5 width=23)
         Hash Cond: (c.customerid = o.customerid)
         ->  Nested Loop  (cost=12738.17..13588.50 rows=166 width=19)
               ->  Subquery Scan f  (cost=12738.17..12745.64 rows=166 width=4)
                     Filter: (f.orank <= 5)
                     ->  Sort  (cost=12738.17..12739.42 rows=498 width=4)
                           Sort Key: ((0)::bigint)
                           ->  HashAggregate  (cost=12710.88..12715.86 rows=498 width=4)
                                 ->  Append  (cost=2029.74..12708.39 rows=498 width=4)
                                       ->  Subquery Scan "*SELECT* 1"  (cost=2029.74..2048.43 rows=298
width=4)
                                             ->  Hash Join  (cost=2029.74..2045.45 rows=298 width=4)
                                                   Hash Cond: ((count(*)) = (max((count(*)))))
```

-> HashAggregate (cost=1007.39..1014.85 rows=597
width=4)
-> Hash Join (cost=733.44..1004.41 rows=597
width=4)
Hash Cond: (o.customerid = c.customerid)
-> Seq Scan on orders o
(cost=0.00..220.00 rows=12000 width=4)
-> Hash (cost=721.00..721.00 rows=995
width=4)
-> Seq Scan on customers c
(cost=0.00..721.00 rows=995 width=4)
Filter: ((country)::text =
'Japan'::text)
-> Hash (cost=1022.34..1022.34 rows=1 width=8)
-> Aggregate (cost=1022.32..1022.33 rows=1
width=8)
-> HashAggregate
(cost=1007.39..1014.85 rows=597 width=4)
-> Hash Join
(cost=733.44..1004.41 rows=597 width=4)
Hash Cond: (o.customerid =
c.customerid)
-> Seq Scan on orders o
(cost=0.00..220.00 rows=12000 width=4)
-> Hash
(cost=721.00..721.00 rows=995 width=4)
-> Seq Scan on
customers c (cost=0.00..721.00 rows=995 width=4)
Filter:
((country)::text = 'Japan'::text)
-> HashAggregate (cost=10655.46..10657.96 rows=200 width=4)
-> Nested Loop (cost=2028.81..10061.45 rows=118803
width=4)
Join Filter: ((count(*)) < o1.numorders)
-> HashAggregate (cost=1007.39..1014.85 rows=597
width=4)
-> Hash Join (cost=733.44..1004.41 rows=597
width=4)
Hash Cond: (o.customerid = c.customerid)
-> Seq Scan on orders o
(cost=0.00..220.00 rows=12000 width=4)
-> Hash (cost=721.00..721.00 rows=995
width=4)
-> Seq Scan on customers c
(cost=0.00..721.00 rows=995 width=4)
Filter: ((country)::text =
'Japan'::text)
-> Materialize (cost=1021.42..1027.39 rows=597
width=8)

-> Subquery Scan o1 (cost=1007.39..1020.82 rows=597 width=8)
-> HashAggregate (cost=1007.39..1014.85 rows=597 width=4)
-> Hash Join (cost=733.44..1004.41 rows=597 width=4)
Hash Cond: (o.customerid = c.customerid)
-> Seq Scan on orders o (cost=0.00..220.00 rows=12000 width=4)
-> Hash (cost=721.00..721.00 rows=995 width=4)
-> Seq Scan on customers c (cost=0.00..721.00 rows=995 width=4)
Filter: ((country)::text = 'Japan'::text)
-> Index Scan using customers_pkey on customers c (cost=0.00..5.06 rows=1 width=15)
Index Cond: (c.customerid = f.customerid)
-> Hash (cost=1020.82..1020.82 rows=597 width=12)
-> Subquery Scan o (cost=1007.39..1020.82 rows=597 width=12)
-> HashAggregate (cost=1007.39..1014.85 rows=597 width=4)
-> Hash Join (cost=733.44..1004.41 rows=597 width=4)
Hash Cond: (o.customerid = c.customerid)
-> Seq Scan on orders o (cost=0.00..220.00 rows=12000 width=4)
-> Hash (cost=721.00..721.00 rows=995 width=4)
-> Seq Scan on customers c (cost=0.00..721.00 rows=995 width=4)
Filter: ((country)::text = 'Japan'::text)
(60 rows)

The total cost is 14629.77
It is much smaller than the cost from part (a)
For query 7.1 the cost of subquery will be taken into consideration every time it is needed, and it will cost a lot of time. But for query 7.2, the view is only created once so is the cost to create it, accessing information by directly using the view can save a lot of time compared to computing it every time.