Patrick Wilson
Zak Pazikowski
EECS 484 – Project 3 – P3answers.pdf

TEC: Total Estimated Cost

Exercise 1:

a)

Solution: 1
Command 1: \di
Then we looked down under indexes and there was one listed (customers_pkey) that was associated with customers
public | categories_pkey | index | pdubslax | categories
public | customers_pkey | index | pdubslax | customers
public | inventory_pkey | index | pdubslax | inventory
public | orders_pkey | index | pdubslax | orders
public | products_pkey | index | pdubslax | products

b)

Solution:

SELECT relpages,relname,relkind FROM pg_class where relkind='r' order by relpages desc limit 2;

returns

relpages	relname	relkind
471	customers	r
385	orderlines	r

SELECT relpages,relname,relkind FROM pg_class where relkind='i' order by relpages desc limit 2;
Returns

relpages	relname	relkind
57	customers_pkey	i
35	orders_pkey	i

c)

Solution:

select attname,n_distinct from pg_stats where tablename='customers';

password	1
state	52
customerid	-1
address2	1
creditcard	-1
age	73
income	5
gender	2
firstname	-1
lastname	-1
address1	-1
city	-1
zip	-0.475
country	11
region	2
email	-1
phone	-1

creditcardtype | 5
creditcardexpiration | 60
username | -1

select count(*) from customers;
20000

so multiply all these values < 1 by 20000 to get #distinct values in each column

password | 1
state | 52
customerid | -1 * 20000 = 20000
address2 | 1
creditcard | -1 * 20000 = 20000
age | 73
income | 5
gender | 2
firstname | -1 * 20000 = 20000
lastname | -1 * 20000 = 20000
address1 | -1 * 20000 = 20000
city | -1 * 20000 = 20000
zip | -0.475 * 20000 = 9500
country | 11
region | 2
email | -1 * 20000 = 20000
phone | -1 * 20000 = 20000
creditcardtype | 5
creditcardexpiration | 60
username | -1 * 20000 = 20000

Two other attributes that you could use as an index to build a b tree would be age or creditcardexpiration since they both have a limited number of distinct values which would allow for clustering and you could see who needs a credit card within the year or who is > 50.

d)

select count(distinct password), count(distinct state), count(distinct customerid), count(distinct address2),count(distinct creditcard),count(distinct age),count(distinct income),count(distinct gender),count(distinct firstname),count(distinct lastname),count(distinct address1),count(distinct city),count(distinct zip),count(distinct country),count(distinct region),count(distinct email),count(distinct phone),count(distinct creditcardtype),count(distinct creditcardexpiration),count(distinct username) from customers;

returns:
1 | 52 | 20000 | 1 | 20000 | 73 | 5 | 2 | 20000 | 20000 | 20000 | 20000 | 9500 | 11 | 2 | 20000 | 20000 | 5 | 60 | 20000

This is exactly equal to the answers derived with the posgres catalog

Attribute Name	Catalog Count	Actual Count
password	1	1
state	52	52
customerid	20000	20000
address2	1	1
creditcard	20000	20000
age	73	73
income	5	5
gender	2	2
firstname	20000	20000
lastname	20000	20000
address1	20000	20000
city	20000	20000

zip	9500	9500
country	11	11
region	2	2
email	20000	20000
phone	20000	20000
creditcardtype	5	5
creditcardexpiration	60	60
username	20000	20000

Excerise 2:

a.

the query:
explain select * from customers where country='Japan';
returns:

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

for an estimated 995 rows.

the query:
select * from customers where country='Japan';
also returns 995 rows, the same as the expected amount.

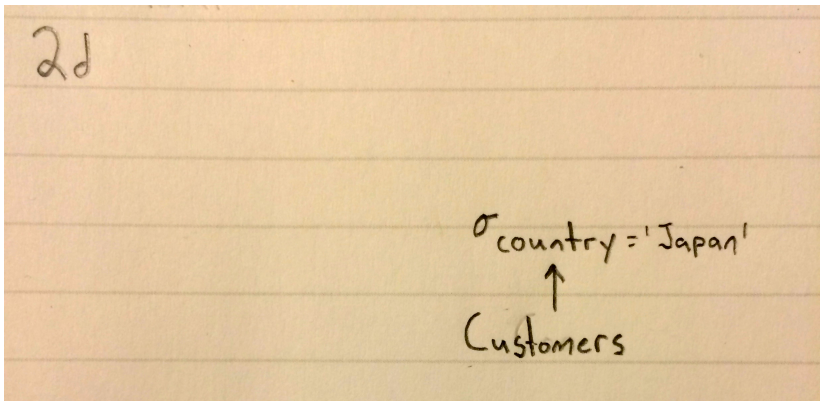
b.
(disk pages read * seq_page_cost) + (rows scanned * cpu_tuple_cost)
where seq_page_cost=1.0 and cpu_tuple_cost=0.01

The query:
SELECT relpages, reltuples FROM pg_class WHERE relname = 'customers';
returns relpages = disk pages read = 471,
reltuples = rows scanned = 20000
Expected execution cost = (471 * 1.0) + (20000 * .01)
Expected execution cost = 671

c.
explain select * from customers where country='Japan';
returns:

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

d.



Exercise 3:

a)
The query:
select relpages from pg_class where relname = 'customers_country';
returns:

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

The customers_country index occupies 59 pages.

b)
The query:
explain select * from customers where country='Japan';
returns:

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

and still has an access method of Sequential Scan, and still has an expected total cost of 721.

c)
The non-clustered index does not lead to any benefits, as it does not actually provide a more efficient way to access the data, and does not usefully presort it. Adding a clause of "where country='Japan'" to the end of the index would increase efficiency.

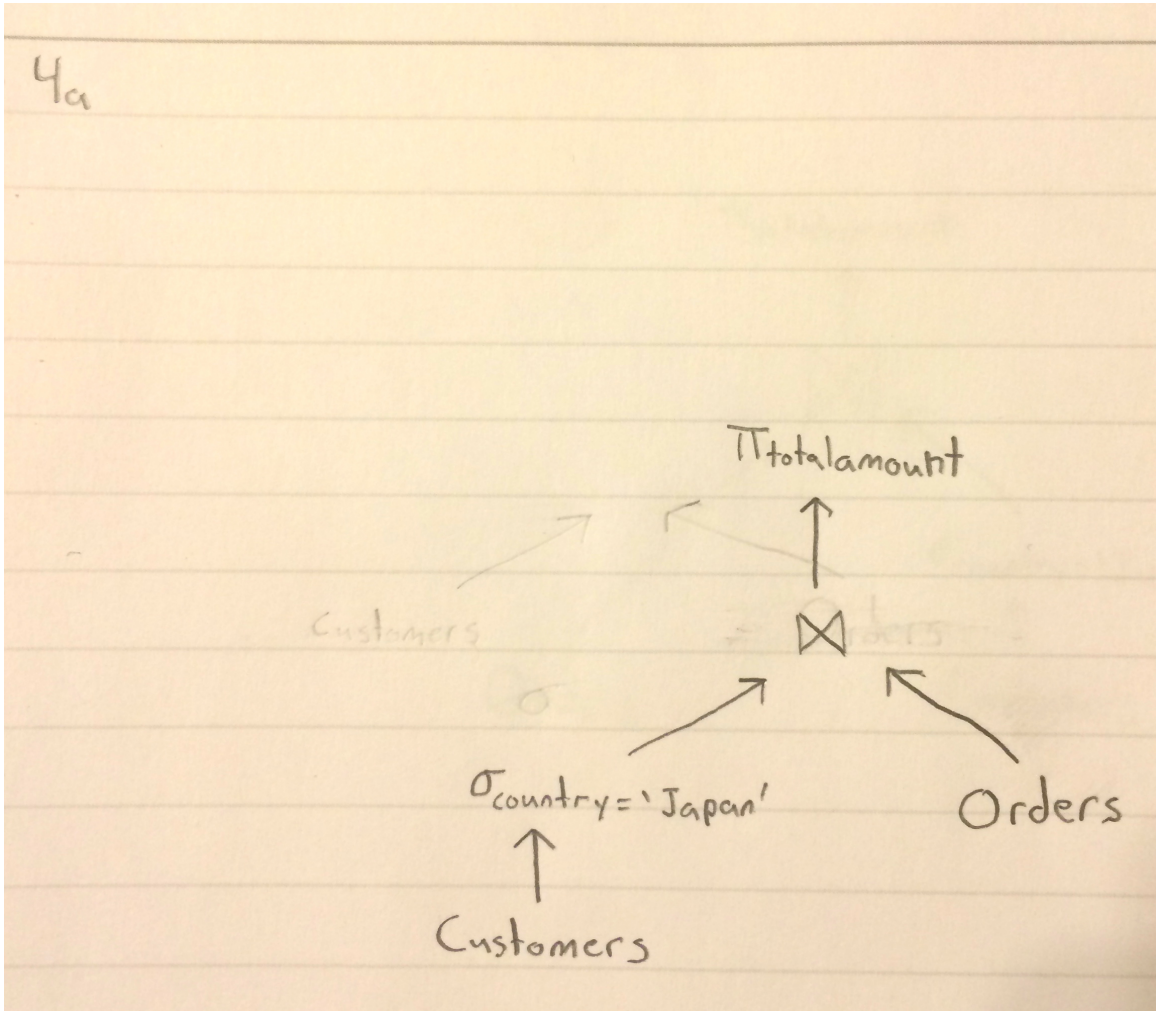
d)
The query:
explain select * from customers where country='Japan';

Now accesses with an Index Scan using customers_country on customers, with an Index condition of ((country)::text = 'Japan'::text).
There is now an estimated cost of 56.66, this is the best plan for these queries, as the estimated cost is 664.34 less than the original cost wihtout clustering.

```
                                  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)
Both queries plan to use scan using coutry='Japan' first, but the clustered query plans to use this as an index condition, whereas the unclustered query plans to use this as a filter.
After, the clustered query will use an Index scan on the customers_country index, and the unclustered query will use a sequential scan.
The clustered query also executes with an estimated cost of 644.34 less than the estimated cost of the unclustered query.

Exercise 4:
a)



b)

```
                          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)
```

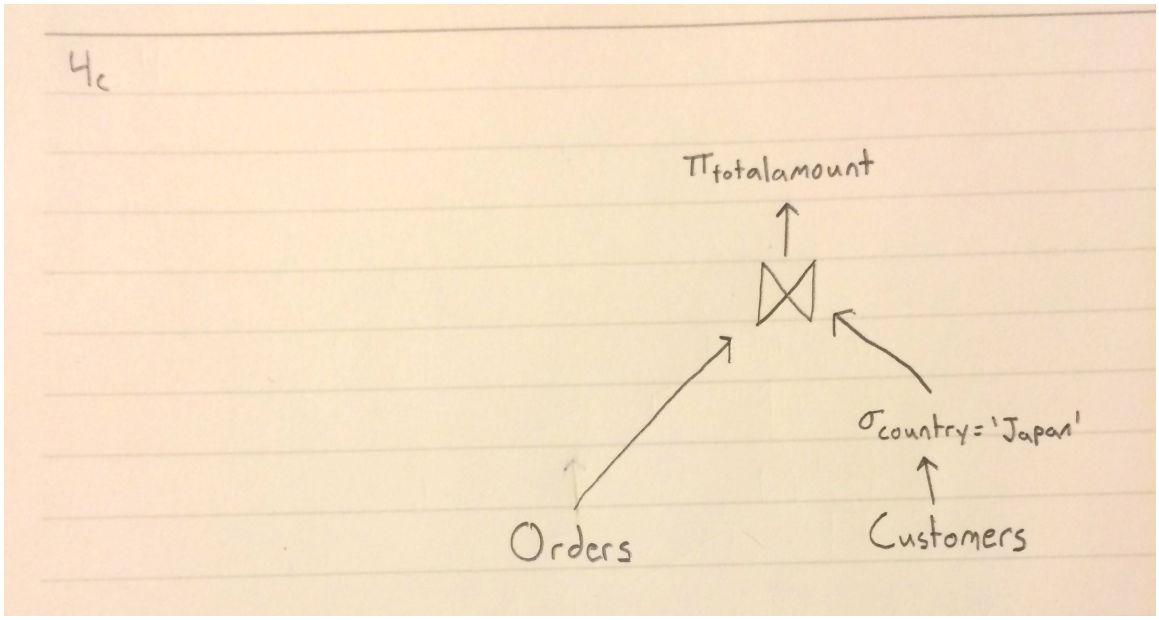
The query optimizer uses a hash join.
The total estimated cost of the hash join is 1004.41.
The estimated cardinality of his query is 597 rows.

c)

```
                          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)
```

With hash join disabled, the query optimizer now uses Merge Join.
The new total estimated cost is 1874.53.



d)

```
                                    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)
```

Once Merge join is disabled, the optimizer uses nested loops.
The estimated cost is 5749.36.

Exercise 5:

a)

```
     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)
```

Hash Join , TEC = 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)
```

Merge Join , TEC = 2325.55

b)

```
                     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)
```

Merge Join , TEC 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)
```

Hash Join , TEC 3813.54

c)Explain why the optimizer may have selected different join algorithms.

Using Merge Join will automatically order on the merge condition. This is required in query 5.2 when we have an order by C.customerid. Hash Join is a faster join and the order condition is not the join condition.

Exercise 6:

```
                       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)
```

a)
The estimated total cost for the query is 5001021.00.

b)
create view OrderCount(customerid, numorders)
as select O.customerId, count(*) from Orders O group by O.customerid;

c)
select C.customerid, C.lastname
from OrderCount OC, Customers C
where OC.numorders > 4 AND OC.customerid = C.customerid;

d)

```
                                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 estimated total cost for my query 6.2 is 1703.29.
This cost is much less than the estimated total cost of query 6.1, as there is no nested query.

Exercise 7:

a)

```
                                          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)
```

TEC = 614927.01

b)

```
create view OrderCountJapan(customerid, numorders)
as select C.customerid, count(*)
from Orders O, Customers C
where C.country = 'Japan' AND O.customerid = C.customerid
group by C.customerid
order by count desc;

create view MoreFrequentJapanCustomers(customerid, oRank) as
select C.customerid, -1 + Rank()
over (
        order by count(*) desc
) as ranks
from Orders O, Customers C
where C.country = 'Japan' AND O.customerid = C.customerid
group by C.customerid
order by count(*) desc;
```

c)

```
select C.customerid,C.lastname, O.numorders
from MoreFrequentJapanCustomers M, OrderCountJapan O, Customers C
where M.oRank<=5 and M.customerid=C.customerid and O.customerid = C.customerid
group by C.customerid,C.lastname, O.numorders
order by count(*) asc;
```

d)

```
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=3080.45..3080.47 rows=6 width=23)
   Sort Key: (count(*))
   ->  HashAggregate  (cost=3080.30..3080.37 rows=6 width=23)
         ->  Hash Join  (cost=2101.18..3080.24 rows=6 width=23)
               Hash Cond: (c.customerid = o.customerid)
               ->  Nested Loop  (cost=1043.87..2007.45 rows=199 width=19)
                     ->  Subquery Scan m  (cost=1043.87..1063.28 rows=199 width=4)
                           Filter: (m.orank <= 5)
                           ->  WindowAgg  (cost=1043.87..1055.81 rows=597 width=4)
                                 ->  Sort  (cost=1043.87..1045.37 rows=597 width=4)
                                       Sort Key: (count(*))
                                       ->  HashAggregate  (cost=1007.39..1016.35 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..4.73 rows=1 width=15)
                           Index Cond: (c.customerid = m.customerid)
               ->  Hash  (cost=1049.84..1049.84 rows=597 width=12)
                     ->  Subquery Scan o  (cost=1042.38..1049.84 rows=597 width=12)
                           ->  Sort  (cost=1042.38..1043.87 rows=597 width=4)
                                 Sort Key: (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)
(31 rows)
```

7.2 TEC : 3080.47
7.1 TEC = 614927.01

This is an example of how extra levels of nesting can greatly increase the cost of a query. When we removed the nested levels and executed the query in a single block, the total cost drastically decreased.