Exercise 1: System Catalog Queries

a) One index is built on the Customers relation.

"customers_pkey" PRIMARY KEY, btree (customerid)

b)

```sql
-- Relations ordered by disk pages occupied
SELECT relname, relpages
FROM pg_class
WHERE relkind = 'r'
ORDER BY relpages desc;

-- Indexes ordered by disk pages occupied
SELECT relname, relpages
FROM pg_class
WHERE relkind = 'i'
ORDER BY relpages desc;
```

Two relations that occupy the most number of disk pages:

customers 471

orderliness 381

Two relations that occupy the most number of disk pages:

customers_pkey 57

orders_pkey 35

c)

```sql
-- Number of distinct values in each column of the Customners table
SELECT attname, n_distinct
FROM pg_stats
WHERE tablename = 'customers'
ORDER by n_distinct;
```

```
       attname        | n_distinct
----------------------+------------
 customerid           |         -1
 firstname            |         -1
 lastname             |         -1
 address1             |         -1
 city                 |         -1
 email                |         -1
 phone                |         -1
 creditcard           |         -1
 username             |         -1
 zip                  |     -0.475
 address2             |          1
 password             |          1
 region               |          2
 gender               |          2
 creditcardtype       |          5
 income               |          5
 country              |         11
 state                |         52
 creditcardexpiration |         60
 age                  |         73
(20 rows)
```

In addition to the attributes that are already indexed, attributes age and creditcardexpiration are suitable for building a B-tree index. They are likely to have range-style queries and sorting.

Also, they have enough many distinct value for the index to be effective. (It is not useful to have an index for an attribute with only two values like gender)

d)

```sql
-- 1d Number of distinct values in each column of Customers table
-- without using the Postgres catalog
SELECT count(distinct customerid) as customerid,
count(distinct firstname) as firstname,
count(distinct lastname) as lastname,
count(distinct address1) as address1,
count(distinct city) as city,
count(distinct email) as email,
count(distinct phone) as phone,
count(distinct creditcard) as creditcard,
count(distinct username) as username,
count(distinct zip) as zip,
count(distinct address2) as address2,
count(distinct password) as password,
count(distinct region) as region,
count(distinct gender) as gender,
count(distinct creditcardtype) as creditcardtype,
count(distinct income) as income,
count(distinct country) as country,
count(distinct state) as state,
count(distinct creditcardexpiration) as creditcardexpiration,
count(distinct age) as age
FROM customers;
```

```
 customerid | firstname | lastname | address1 | city  | email | phone | creditcard | username | zip
------------+-----------+----------+----------+-------+-------+-------+------------+----------+------
      20000 |     20000 |    20000 |    20000 | 20000 | 20000 | 20000 |      20000 |    20000 | 9500

 address2 | password | region | gender | creditcardtype | income | country | state | creditcardexpiration | age
----------+----------+--------+--------+----------------+--------+---------+-------+----------------------+-----
        1 |        1 |      2 |      2 |              5 |      5 |      11 |    52 |                   60 |  73
```

Table comparing expected count from the catalog compare with the actual count values

column name	n_distinct	expected count	actual count
customerid	-1	20000	20000
firstname	-1	20000	20000
lastname	-1	20000	20000
address1	-1	20000	20000
city	-1	20000	20000
email	-1	20000	20000
phone	-1	20000	20000
creditcard	-1	20000	20000
username	-1	20000	20000
zip	-0.475	9500	9500
address2	1	1	1
password	1	1	1
region	2	2	2
gender	2	2	2
creditcardtype	5	5	5
income	5	5	5

country	11	11	11
state	52	52	52
creditcardexpiration	60	60	60
age	73	73	73

The expected count from the catalog is the same with the actual count values.

Exercise 2: Equality Query

a)

```
tywei=> EXPLAIN SELECT * FROM customers WHERE country = 'Japan';
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on customers  (cost=0.00..721.00 rows=995 width=156)
   Filter: ((country)::text = 'Japan'::text)
(2 rows)
```

The estimated cardinality of this query is 995.

```
tywei=> SELECT count(*) FROM customers WHERE country = 'Japan';
 count
-------
   995
(1 row)
```

It is the same as the actual value.

b)

(disk pages read * seq_page_cost) + (rows scanned * cpu_tuple_cost) + (rows scanned * cpu_operator_cost)

= (471 * 1.0) + (20000 * 0.01) + (20000 * 0.0025)

= 721

c)

$\sigma_{country\ ='Japan'}(customers)$

$\sigma_{country\ ='Japan'}$

|

CUSTOMERS

Exercise 3: Equality Query with Indexes

a)

```sql
-- 3a Disk page occupied by index customers_country
SELECT relname, relpages
FROM pg_class
WHERE relname = 'customers_country';
```

```
      relname      | relpages
-------------------+----------
 customers_country |       59
(1 row)
```

The index customers_country occupies 59 pages.

b)

```
tywei=> EXPLAIN SELECT * FROM customers WHERE country = 'Japan';
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on customers  (cost=0.00..721.00 rows=995 width=156)
   Filter: ((country)::text = 'Japan'::text)
(2 rows)
```

The query optimizer checks the condition for each row it scans, and outputs only the ones that pass the condition. The estimated total cost of executing the best plan for this query is 721.00.

c) The estimated cost of non-clustered index is the same as using no index as in Exercise 2. The reason may be that lots of rows pass the condition. Since the index is non-clustered, number of disk pages read using the index is similar to using no index (sequential scanning). Besides, using index takes addition cost to retrieve the object.

d)

```
tywei=> EXPLAIN SELECT * FROM customers WHERE country = 'Japan';
                                       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)
```

The query optimizer fetches the rows using the index customers_country. The estimated total cost of executing the best plan for this query is 56.66.

e) The best plan of clustered index cost much smaller than the best plan of non-clustered index for the query. Since the index is clustered, the number of pages needed to read is much smaller than non-clustered index.

Exercise 4: Join Query

a)

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

$\pi_{totalamount}$

|

$\bowtie$

customerid = customerid

$\pi_{customerid}$ $\quad\quad\quad\quad$ $\pi_{customerid,\ totalamount}$

| $\quad\quad\quad\quad$ |

$\sigma_{country='Japan'}$ $\quad\quad\quad\quad$ ORDER

|

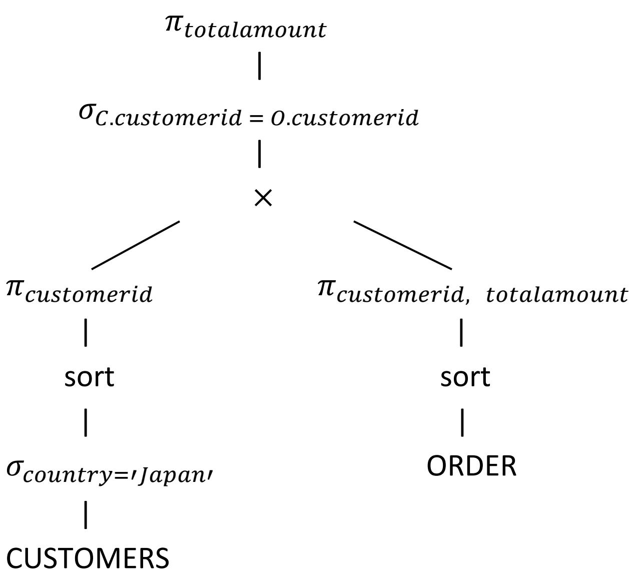
CUSTOMERS

b) The query optimizer uses hash join. The total estimated cost is 1004.41. The estimated result cardinality of this query is 597.

c)

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

Merge join is used by the query optimizer now. The total estimated cost is 1874.53.



d)

```
tywei=> SET enable_mergejoin = off;
SET
tywei=> SET enable_hashjoin = off;
SET
tywei=> EXPLAIN SELECT totalamount
tywei-> FROM Customers C, Orders O
tywei-> WHERE C.customerid=O.customerid AND C.country = 'Japan';
                                        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)
```

Nested loop is used now. The estimated total cost is 5749.36.

Exercise 5

a)

```
-- 5a.first selected join
EXPLAIN SELECT AVG(totalamount) as avgOrder, country
    FROM Customers C, Orders O
    WHERE C.customerid = O.customerid
    GROUP BY country
    ORDER BY avgOrder;
```

The query optimizer selects hash join for Query 5.1. The total expected cost is 1501.36.

```
-- 5a.second selected join
SET enable_hashjoin = off;
EXPLAIN SELECT AVG(totalamount) as avgOrder, country
    FROM Customers C, Orders O
    WHERE C.customerid = O.customerid
    GROUP BY country
    ORDER BY avgOrder;
```

Disable the selected join algorithm, merge join is selected. The total expected cost is 2325.55.

b)

```
-- 5b.first selected join in Query 5.2
SET enable_hashjoin = on;
EXPLAIN SELECT *
    FROM Customers C, Orders O
    WHERE C.customerid = O.customerid
    ORDER BY C.customerid;
```

The query optimizer selectes merge join for Query 5.2. The total expected cost is 2265.19.

```
--5b.second selected jonin in Query 5.2
SET enable_mergejoin = off;
EXPLAIN SELECT *
    FROM Customers C, Orders O
    WHERE C.customerid = O.customerid
    ORDER BY C.customerid;
```

Disable the selected join algorithm, hash join is selected. The total expected cost is 3813.54.

c)

The optimizer selected different join algorithms for two queries. For Query 5.1, sorting on joining attributes is not required, so hash table is cheaper when joining large tables. For Query 5.2, sorting on joining attributes is required anyway due to the ‘order by’ clause, and thus merge sort is preferred.

Exercise 6

a)

```
--6a.best plan for Query 6.1
EXPLAIN SELECT C.customerid, C.lastname
    FROM Customers C
    WHERE 4 < (SELECT COUNT(*)
        FROM Orders O
        WHERE O.customerid = C.customerid);
```

The estimated total cost of best plan is 5001021.00.

b)

```
--6b.query to create a view OrderCount
CREATE VIEW OrderCount(customerid, numorders)
    AS SELECT C.customerid, COUNT(*)
    FROM Customers C, Orders O
    WHERE O.customerid = C.customerid
    GROUP BY C.customerid;
```

c)

```
--6c.Query 6.2
SELECT C.customerid, C.lastname
    FROM Customers C, OrderCount  OC
    WHERE OC.customerid = C.customerid
    AND OC.numorders > 4;
```

d)

```
--6d.Query 6.2 evaluation
EXPLAIN SELECT C.customerid, C.lastname
    FROM Customers C, OrderCount  OC
    WHERE OC.customerid = C.customerid
    AND OC.numorders > 4;
```

The estimated total cost for Query 6.2 is 3887.44. It’s much quicker than nested query in (a).

Exercise 7

a)

```
--7a.best plan for Query 7.1
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;
```

The estimated total cost of best plan is 614927.01.

b)

```sql
--7b.query to create a view MoreFrequentJapanCustomers
CREATE VIEW OrderCount(customerid, numorders)
    AS SELECT C.customerid, COUNT(*)
    FROM Customers C, Orders O
    WHERE O.customerid = C.customerid
    AND C.country = 'Japan'
    GROUP BY C.customerid;
CREATE VIEW MoreFrequentJapanCustomers(customerid, oRank)
    AS SELECT O1.customerid, COUNT(*)
    FROM OrderCount O1
    LEFT OUTER JOIN OrderCount O2
    ON O1.numorders < O2.numorders
    GROUP BY O1.customerid;
```

c)

```sql
--7c.Query 7.2
SELECT C.customerid, C.lastname, OC.numorders
    FROM Customers C, OrderCount OC, MoreFrequentJapanCustomers M
    WHERE OC.customerid = C.customerid
    AND C.customerid = M.customerid
    AND M.oRank <= 5
    ORDER BY C.customerid;
```

d)

```sql
--7d.Query 7.2 evaluation
EXPLAIN SELECT C.customerid, C.lastname, OC.numorders
    FROM Customers C, OrderCount OC, MoreFrequentJapanCustomers M
    WHERE OC.customerid = C.customerid
    AND C.customerid = M.customerid
    AND M.oRank <= 5
    ORDER BY C.customerid;
```

The estimated total cost for Query 7.2 is 12923.09. It's much quicker than nested query in (a). In (a), the sub-query "5 >= …"is evaluated once for each row in outer query ORDERCOUNTS1.numorders, which is inefficient.