Exercise 1: System Catalog Queries

a)
\d+ customers
1. "customers_pkey".

b)
(The queries are too many to list, so we list them in the P3queries.sql)
Relation: customers(471 disk pages), orderlines(385 disk pages).
Index: customers_pkey(57 disk pages), orders_pkey(35 disk pages).

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

attname	n_distinct
customerid	20,000
firstname	20,000
lastname	20,000
address1	20,000
address2	20,000
city	20,000
state	52
zip	9,500
country	11
region	2
email	20,000
phone	20,000
creditcardtype	5
creditcard	20,000
creditcardexpiration	60
username	20,000
password	20,000
age	73
income	5
gender	2

Because B-tree is good for range-style sorting and queries, we can choose the attributes with distinct value for every data. Also we want to avoid publishing users' private information. In view of this, email and username are good choice for B-tree index.

d)
(The queries are too many to list, so we list them on the P3queries.sql)

attname	n_distinct (postgres)	count(*) (query)
customerid	20,000	20,000
firstname	20,000	20,000
lastname	20,000	20,000
address1	20,000	20,000
address2	20,000	20,000
city	20,000	20,000
state	52	52
zip	9,500	9,500
country	11	11
region	2	2
email	20,000	20,000
phone	20,000	20,000
creditcardtype	5	5
creditcard	20,000	20,000
creditcardexpiration	60	60
username	20,000	20,000
password	20,000	20,000
age	73	73
income	5	5
gender	2	2

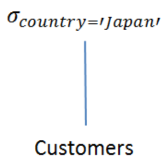
The expected counts from the catalog are all the same with the actual count values.

Exercise 2: Equality Query

a)
Explain select * from customers where country = 'Japan';
The estimated cardinality is 995 rows while the actual value derived from sql is also 995.

b)
The estimated cost is computed as (disk pages read * seq_page_cost) + (rows scanned * cpu_tuple_cost) + (row scanned * cpu_operator_cost).
471*1 + 20,000*0.01 + 20,000*0.0025 = 721.

c)



Exercise 3: Equality Query with Indexes

a)
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'customers%';

customers_country occupies 59 pages.

b)
EXPLAIN select * from customers where country = 'Japan';

The query optimizer selects sequential scan.
The estimated total cost = 721.00

c)
The customers_country is not clustered, so the accesses might read in different pages that takes a lot of loading time. So the non-clustered index not leads to significant benefits.

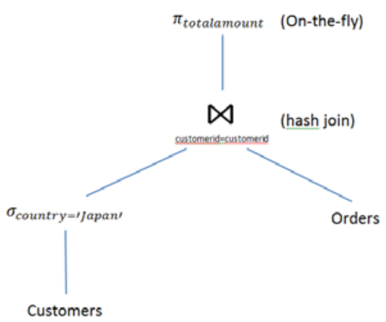
d)
EXPLAIN select * from customers where country = 'Japan';

The query optimizer selects index scan.
The estimated total cost = 56.66.

e)
The best plan of clustered index is much faster than the best plan of non-clustered index. The clustered index is sorted by the order of customers data so that the access can more efficient. However, the sequential accesses in bitmap method might read in different pages and takes a lot of time.

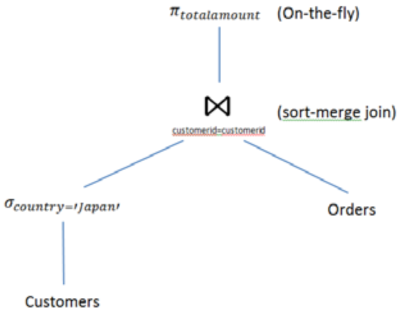
Exercise 4: Join Query

a)



b)
The query optimizer uses has join.
The total estimated cost = 1004.41.
The estimated result cardinality = 597 rows.

c)
The query optimizer uses merge join now.
The total estimated cost = 1874.53.



d)
The query optimizer uses nested loop now.
The total estimated cost = 5749.36.

Exercise 5: Join Selection

a)
The query optimizer first selects hash join and the total expected cost = 1501.36.
After disable the hash join algorithm, it selects merge join and the expected total cost = 2325.55.

b)
The query optimizer first selects merge join and the total expected cost = 2265.19.
After disable the merge join algorithm, it selects hash join and the expected total cost = 3813.54.

c)
Hash join is much more efficient for table with no index and can run the query parallel. In query 5.1, we want data sort by avgOrder which has no index on this. As a result, the optimizer choose hash join.
On the contrast, since the query 5.2 orders data by customerid, which contains the index of the dataset, the cost of sort merge join can be lowered dramatically.

Exercise 6: Correlated Subquery

a)
The estimated total cost = 5001021.

b)
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)

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

d)

The estimated total cost = 3887.44. It much lower than the estimated cost for the nested query from part(a).

<u>Exercise 7: Query Optimization</u>

a)

The estimated total cost = 614927.01.

b)

```
CREATE VIEW OrderCountJapan(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 OCJ1.customerid, count(*)
FROM OrderCountJapan OCJ1 LEFT JOIN OrderCountJapan OCJ2
ON OCJ1.numorders < OCJ2.numorders
GROUP BY OCJ1.customerid;
```

c)

```
SELECT C.customerid, C.lastname, MFJC.oRank, OCJ.numorders
FROM MoreFrequentJapanCustomers MFJC, Customers C, OrderCountJapan OCJ
WHERE MFJC.customerid = C.customerid AND MFJC.oRank <=5 AND
C.customerid = OCJ.customerid
ORDER BY C.customerid;
```

d)

The estimated total cost = 12952.01.

It's much lower than the answer from part (a) because in part(a), the sub-query is evaluated once for each row processed by the outer query so that we need to join every time the same two tables when we access the inner query. However, the query 7.2 create two view to store the result of the join of tables so that we can directly use the result instead of generate them every time, and it would save a lot of processing time so the query 7.2 is much more efficient than query 7.1.