Problem Set #2

1. Suppose an important customer says "It is essential that your DBMS processes join predicates of the form (A.x = B.x or A.y=B.y) quickly". Normally, you would say: tough beans. But the survival of your DBMS company is dependent on this customer using your product. Your DBMS supports only nested loops and hash join algorithms, and uses the System-R algorithm for generating physical access plans. You can't make major changes to your DBMS query optimization subsystem. Your manager has an insight – generalize the hash join algorithm. But how?

2. Another important customer says "it is essential that your DBMS processes join predicates of the form (A.x>B.x) quickly". As before, you'd rather say: tough beans. But your manager asked you again to perform another miracle. Using the same constraints as before, how would you save your company? Hint: think alternative main-memory data structures.

3. Consider the following nested SQL query, which says retrieve x values from each A tuple where q=40 and there are **no** tuples in C that could join with attribute w of that tuple in A:

```
select A.x
from A
where A.q=40 and not exists (
    select *
    from C
    where C.w = A.w )
```

   a. Use a Kim or magic set rewrite of this query as a sequence of 2 non-nested queries. Hint: SQL minus and select-into operations.
   b. Rewrite this query as a single unnested query. Hint: outerjoins.

4. What does this query mean? (This query could be executed on the database of Project 1).

```
select pname
from parts
where not exists (
        select cname
        from customers natural join zipcodes
        where city = 'Austin' and not exists (
                select ono
                from orders natural join odetails
                where cno = customers.cno
                        and parts.pno = pno )    )
```

Problem Set #2 – Solutions

1. Suppose an important customer says "It is essential that your DBMS processes join predicates of the form (A.x = B.x or A.y=B.y) quickly". Normally, you would say: tough beans. But the survival of your DBMS company is dependent on this customer using your product. Your DBMS supports only nested loops and hash join algorithms, and uses the System-R algorithm for generating physical access plans. You can't make major changes to your DBMS query optimization subsystem. Your manager has an insight – generalize the hash join algorithm? If so, how?

Answer: there are a variety of ways to solve this problem. The absolute simplest is to add another join processing algorithm that deals with OR join predicates, following the advice of your manager. The basic idea is to create **2** hash tables for a hash join, one for attribute **x** and another for attribute **y**. For each outer loop record $a \in A$, you find via one hash table all records of B that join with $a.x$, and via the second, all records of B that join with $a.y$. The problem here is that you may produce duplicate (a, b) record pairs. You will have to sort the output of your "OR-join" algorithm and remove duplicates.

2. Another important customer says "it is essential that your DBMS processes join predicates of the form (A.x>B.x) quickly". As before, you'd rather say: tough beans. But your manager asked you again to perform another miracle. How would you save your company?

Answer: add another algorithm, similar to hash joins. Instead of inhaling all records from the outer stream and storing them in a hash structure, store them in a **binary tree** (or some structure that maintains order so that you can perform efficient record retrievals for queries like key>value). Then read each record $b \in B$, find all records in A such that $A.x > b.x$.

3. Consider the following nested SQL query, which says retrieve (x,y) pairs where q=40 and A tuples join with B via column z and there are **no** tuples in C that could join with the current tuple in A:

```
select A.x
from A
where A.q=40 and not exists (
    select *
    from C
    where C.w = A.w )
```

   a. Use a Kim or magic set rewrite of this query as a sequence of 2 non-nested queries. Hint: SQL minus and select-into operations.

   Answer: my first instinct for not exists is to create the set of values for which desired A tuples WILL be selected by a semi-join.

```
select w
into goodOnes
from (select w from A) minus (select w from C)
```

   Followed by a nice, simple select

```
Select A.x
from A, goodOnes
where A.q=40 and A.w=goodOnes.w
```

   It is possible to move the A.q=40 predicate into the goodOnes predicate (e.g. **(select w from A where q=40)**) for further optimization.

   b. Rewrite this query as a single unnested query. Hint: outerjoins.

   Answer: again, this requires some creativity. You can use a right outerjoin to produce a useful table:

```
select w
from A right outer join C
```

   Where the tuples of interest have C.w!=null.

   Now you can use this to rewrite the query as

```
select A.x
from A right outer join C
where A.q=40 and C.w!=null
```

4 What does this query mean? (This query could be executed on the database of Project 1).

```
select pname
from parts
where not exists (
        select cname
        from customers natural join zipcodes
        where city = 'Austin' and not exists (
                select ono
                from orders natural join odetails
                where cno = customers.cno
                        and parts.pno = pno )    )
```

Answer: what are the names of parts that have been ordered by all customers in Austin?