Exercises 09

Query Optimisation and Execution

1. Consider the following tables relating to trips on a suburban bus network

```
Trip(fromPlace:integer, toPlace:integer, tripDate:date)
Place(placeId:integer, address:string, suburb:string)
```

a. Write an SQL query that returns all of the addresses in Randwick that are the destination of a trip on March 4, 2005.

Answer:

```
select address
from   Trip, Place
where  tripDate='04-03-2005' and suburb='Randwick' and toPlace=placeId;
```

b. Give a naive translation of the SQL query into a relational algebra expression.

Answer:

```
Temp1  = Trip Join[toPlace=placeId] Place
Temp2  = Sel[tripDate='04-03-2005' and suburb='Randwick'](Temp1)
Result = Proj[address](Temp2)
```

c. Translate the naive relational algebra expression into an equivalent expression using pushing of selections and projections.

Answer:

```
TT     = Proj[toPlace](Sel[tripDate='04-03-2005'](Trip))
PP     = Proj[placeId,address](Sel[suburb='Randwick'](Place))
Result = Proj[address](TT Join[toPlace=placeId] PP)
```

d. Translate the optimized relational algebra expression into the most directly corresponding SQL query.

Answer:

```
select address
from   (select toPlace from Trip where [tripDate='04-03-2005') TT,
       (select placeId,address from Place where suburb='Randwick') PP
where  toPlace = placeId;
```

2. What are the possible join trees (without cross-products) for each of the following queries:

a. `select * from R,S,T where R.a=S.b and S.c=T.d`

Answer:

((R join S) join T)   or   (R join (S join T))

b. `select * from R,S,T where R.a=S.b and T.c=R.d`

Answer:

((R join S) join T)   or   ((R join T) join S)

c. `select * from R,S,T where R.a=S.b and S.c=T.d and T.e=R.f`

Answer:

any of the above

d. `select * from R,S,T,U`
`where R.a=S.b and S.c=T.d and T.e=R.f and T.g=U.h and S.i=U.j`

Answer:

((R join S) join (T join U)) or (((R join S) join T) join U) or (R join (S join (T join U)) or ...

(pretty much any order that does not involve a direct join of R and U)

Do not include trees/sub-trees that are reflections of other tree/subtrees.

3. Consider a table *R(a,b,c)* and assume that

- all attributes have uniform distribution of data values
- attributes are independent of each other
- all attributes are `NOT NULL`
- *r = 1000*, *V(a,R) = 10*, *V(b,R) = 100*

Calculate the expected number of tuples in the result of each of the following queries:

a. `select * from R where not a=`*k*
b. `select * from R where a=`*k* `and b=`*j*
c. `select * from R where a in (`*k*`,`*l*`,`*m*`,`*n*`)`

where *j*, *k*, *l*, *m*, *n* are constants.

Answer:

a. `select * from R where not a=`*k*

Since the set of values for each attribute is uniformly distributed, and since there are 10 possible values for the `a` attribute, we would expect that 1/10 of the tuples would satisfy `a=`*k*. This means that 90% of the tuples would *not* satisfy the condition. Since there are 1000 tuples, we would expect 900 of them to be in the result of the query.

An alternative formulation of the above

- *Prob(a=k)* = 0.1
- *Prob(a!=k)* = 1 - *Prob(a=k)* = 0.9
- #result tuples = *r* × *Prob(a!=k)* = 1000 × 0.9 = 900

b. `select * from R where a=`*k* `and b=`*j*

If attributes are independent, then we'd expect that the chance of attribute `R.a` having a specific value was 1/10 and the chance of attribute `R.b` then have a particular value was 1/100 of that (i.e. the likelihoods multiply).

Using the probability-based formulation:

- *Prob(a=k)* = 0.1, Prob(b=j) = 0.01
- *Prob(A=k and B=j)* = *Prob(A=k)* × *Prob(B=j)* = 0.001
- #result tuples = *r* × *Prob(A=k and B=j)* = 1000 × 0.001 = 1

c. `select * from R where a in (`*k*`,`*l*`,`*m*`,`*n*`)`

If we have alternatives for possible values for attribute `R.a` then we simply sum up the chances for any one of the values to occur.

Using the probability-based formulation:

- *Prob(a=x)* = 0.1, where *x* is any value in *domain(a)*
- *Prob(a∈{x,y,...})* = *Prob(a=x)* + *Prob(a=y)* + ...
- *Prob(a in (k,l,m,n))* = 4 × *Prob(a=k)* (assuming uniform distribution)

- #result tuples = *r* × *Prob(a in (k,l,m,n))* = 1000 × 0.4 = 400

4. Consider the following tables relating to retail sales:

```
create table Item (
    iname       text,
    category    text,
    primary key (name)
);
create table Store (
    sname       text,
    city        text,
    street      text,
    primary key (city,street)
);
create table Transaction (
    item        text references Item(iname),
    store       text references Store(sname),
    tdate       date,
    primary key (item,store,tdate)
);
```

Consider the following query (expressed as SQL and relational algebra):

```
select category,city
from   Item, Store, Transaction
where  iname=item and store=sname and
       tdate='20-12-2004' and city='Sydney';

JoinResult   = Item Join[iname=item] Transaction Join[store=sname] Store
SelectResult = Sel[tdate='20-12-2004' and city='Sydney'](JoinResult)
FinalResult  = Proj[category,city](SelectResult)
```

Show the three "most promising" relational algebra expressions that the query optimizer is likely to consider; then find the most efficient query plan and estimate its cost.

Assume 50 buffer pages and the following statistics and indices:

- `Item`: 50,000 tuples, 10 tuples/page.
  Indexing: hashed on `iname` (assume no overflows).
- `Store`: 1,000 tuples, 5 tuples/page; 100 cities.
  Index1: Unclustered hash index on `sname`. Index2: Clustered 2-level B+tree on `city`.
- `Transaction`: 500,000 tuples, 25 tuples/page; 10 items bought per store per day.
  The relation stores transactions committed over a 50 day period.
  Index: 2-level clustered B+tree on the pair of attributes `store,ttime`.

Answer:

The three most promising relation algebra expressions:

```
Exp1:
    Temp1    = Sel[tdate='20-12-2004'](Transaction)
    Temp2    = Sel[city='Sydney'](Store)
    Temp3    = Temp1 Join[store=sname] Temp2
    Temp4    = Item Join[iname=item] Temp3
    Result   = Proj[category,city](Temp4)

Exp2:
    Temp1    = Sel[city='Sydney'](Store)
    Temp2    = Transaction Join[store=sname] Temp1
    Temp3    = Sel[tdate='20-12-2004'](Temp2)
    Temp4    = Item Join[iname=item] Temp3
    Result   = Proj[category,city](Temp4)
```

```
Exp2:
    Temp1   = Sel[tdate='20-12-2004'](Transaction)
    Temp2   = Temp1 Join[store=sname] Store
    Temp3   = Sel[city='Sydney'](Temp2)
    Temp4   = Item Join[iname=item] Temp3
    Result  = Proj[category,city](Temp4)
```

The most efficient relational algebra expression is a variation on `Exp2`:

```
Best:
    Temp1   = Sel[city='Sydney'](Store)
    Temp2   = Temp1 Join[sname=store] Transaction
    Temp3   = Sel[tdate='20-12-2004'](Temp2)
    Temp4   = Temp3 Join[item=iname] Item
    Result  = Proj[category,city](Temp4)
```

Costs for the various operations (assuming pipelining, so no writes):

a. Cost of `Temp1`: 2 + 2 = 4
   Assume 10 tuples/city (uniform distribution). Since there are 5 tuples/page and the file is sorted on `city`, these 10 Sydney tuples should appear in 2 pages (worst-case 3 pages). Cost of traversing B+tree to find the first matching tuple is 2 (since tree has depth 2), plus cost of reading 2 data pages, gives a total of 4 page reads.

b. Cost of `Temp2+Temp3`: 10 * 3 = 30
   Use indexed nested loop join and combine it with the selection on `tdate` as follows: the index on `Transaction` is on the pair of attributes (`store,tdate`); the join condition involves just `store`; however, the selection also gives us the `tdate` value; so, for each `store`, we can form an index key (`store,tdate`) and do an index lookup. Since there are 10 tuples in the result of `Temp1`, we will perform 10 index lookups on `Transaction`. Each index lookup will yield 10 results (we are given that 10 items are sold in each store on each day). Since the `Transaction` data file is sorted (clustered) by `store` (then `tdate`), and there are 25 tuples per page, all 10 tuples will likely be on a single page (worst case 2 pages). The B+tree index is depth 2, so there are 2 index page reads and 1 data page read for each index lookup (i.e. 3 page reads). Since we perform 10 index lookups, this gives a total of 30 reads, which produces 100 tuples.

c. Cost of `Temp4`: 100 * 1 = 100
   The join between `Temp3` and `Item` can also use an index nested loop join, based on the fact that `Item` is hashed on the `iname` value, and each tuple from `Temp3` gives us an `item` value to use as a hash key. For each of the 100 tuples from `Temp3`, we do a hash access to the page containing the matching tuple (there will be only 1 matching tuple, since `iname` is a key). This requires exactly 100 page reads.

Total cost is the sum of these: 4 + 30 + 100 = 134 page reads.

Note that we ignore the cost of the projection in the above calculation. We can do this because we generally ignore the cost of writing results (especially in the context of comparing relational algebra expressions for a given query, since all expressions produce the same result). Also, we don't need to worry about filtering duplicates in this case, since the query didn't ask for it (i.e. not `select distinct`).