CS532 : Homework 4

Name : Shriram Ananda Suryawanshi (**CS532**)
B-Number: B00734421 (Spring 2018)

CS532 : Homework 4

Name - Shriram Ananda Suryawanshi (**CS532**)
B-Number - B00734421 (Spring 2018)

---

1. **Consider the following SQL query for a banking database system. For simplicity, assume that there is only one account type. Also, the status of a customer can be bronze, silver, or gold member.**

```
select account_id, customer_name
from accounts
where status = 'gold member' and customer_city = 'Vestal';
```

Suppose that (a) each account tuple occupies 200 bytes; (b) all pages are of size 4KB (i.e., 4000 bytes); (c) there are 50 customers (out of total 10,000 customers) with gold member status; and (d) 1000 customers live in Vestal. Discuss how to evaluate the query for the following cases (Hint: If there are different options for a case, try to minimize the number of page I/Os. Also, don't forget to differentiate sequential I/Os from random I/Os):

   a. **Case 1: There is no index on either status or customer_city.**

   - In this case, we need to scan each tuple in the table 'accounts' and check if the required conditions are met. If yes, return that tuple and search for the next until the last tuple.

   b. **Case 2: There is an index on customer_city but no index on status.**

   - As we have index on customer_city, first find all the tuples with customer_city ='Vestal', and perform sequential search on the resultant tuples to find only those who has gold membership.

   c. **Case 3: There is a secondary index on status and a secondary index on customer_city.**

   - Considering the fact that number of people live in Vestal are far more than the number of people having gold membership at the bank.
   Hence, the best approach would be, find all the tuples who has status as gold member, and then among those tuples find only those tuples who has city Vestal.

   d. **Case 4: There is a primary index on customer_city and a secondary index on status.**

   - From the details given, each tuple occupies 200 bytes and each page is of 4000 bytes. That means, each page will hold 20 tuples. Also, we have total 10,000 tuples, that means we have 500 pages for the whole table.
   Now, we have primary index on city, and number of tuples having Vestal as city are 1000. That means, we have 50 pages containing tuples with city Vestal. And, as this is indexed, pages are consecutive.
   Also, we have secondary index on status and we have 50 tuples having gold membership. But, the probability of tuples residing on consecutive pages satisfying gold member conditions is very less.
   Hence, the best approach would be to use primary index to find all the tuples having Vestal as city and in the resultant tuples look for the tuples having gold membership.

2. **Consider the join $R \infty_{R.A = S.B} S$, where R and S are two relations. Three join methods, i.e., nested loop, sort merge and hash join, are discussed in class. Nested loop and sort merge may benefit from the existence of indexes. Identify three different situations (i.e., with given sizes of R and S, and the index status on R.A and/or S.B) such that each of the following claims is true for one situation.**
Here the performance is compared based on the number of I/O pages. Suppose N and M are sizes of R and S in pages, respectively. Without loss of generality, assume that N > M. Also, assume that the memory buffer for the join is not large enough to hold the entire R. Justify your answer.

   a. **Nested loop outperforms sort merge and hash join.**

      - Nested loop outperforms sort merge and hash join, when one of the relation participating in the join has primary index on the join condition and other relation is considerably less in size.
      Let's consider that relation S contains very small number of tuples, while relation R has primary index on the attribute A. Assume that we have page size that contains 20 tuples for both relations, and relation S has 20 tuples i.e. 1 page and relation R has 2000 tuples i.e. 100 pages.
      If we consider that B+ tree used for indexing, and it has two levels for attribute A, we have nested loop approach cost approximately 20 * (2 + 1) = 60 pages of I/O.
      In the case of sort merge and hash join, we have to bare the cost of minimum of 1 + 100 = 101 page I/O.
      Hence, in such case, nested loop is the best approach.

   b. **Sort merge outperforms nested loop and hash join.**

      - Sort merge outperforms nested loop and has join, when both relations are already sorted on their respective joining attribute and there are no repeating values under at least one joining attribute. Also, sort-merge performs well when both R and S are large relations.
      In such cases, we need only N + M page I/O with sort merge.
      In the case of nested loop and has join, we cannot keep the any of the relation in the memory as they are of large size. Hence, one of the relation has to be traversed multiple times resulting into page I/O more than N + M.
      Hence, in such cases, sort merge would be the best approach.

   c. **Hash join outperforms sort merge.**

      - Hash join outperforms the sort merge, when we have equijoin, and hash table on the smaller relation can be held in a memory, leaving at least one page space in the memory buffer for other relation. Here, the larger table cannot be held into memory buffer and it is not sorted, and there are no indexes on attributes participating in the joins.
      In such a case, hash join needs at least N + M page I/O, while sort merge needs to sort the relations first, hence it would require more than N + M page I/O.
      Hence, in such cases, hash join outperforms sort merge.

3. **Consider the following query execution plans:**

Plan A: σ GPA ≥ 2 (Students $\bowtie_{Students.SSN = Faculty.SSN}$ Faculty)

Plan B: (σ GPA ≥ 2 (Students)) $\bowtie_{Students.SSN = Faculty.SSN}$ Faculty

Which plan will be selected if the four rules for query optimization in Chapter 11 are applied? Is the selected plan more efficient than the other one? Justify your answers.

- As per the rule 1, we should perform selections as early as possible. By this rule, we must follow the Plan B, as it is selecting the students having GPA greater than or equal to 2 first and then performs the join operation.
But, if we analyze the plans considering the tuples both relations having, there will be very few students who belongs to faculty as well. Hence, the Plan A will be more efficient in this case.

4. **Consider the following three relations:**

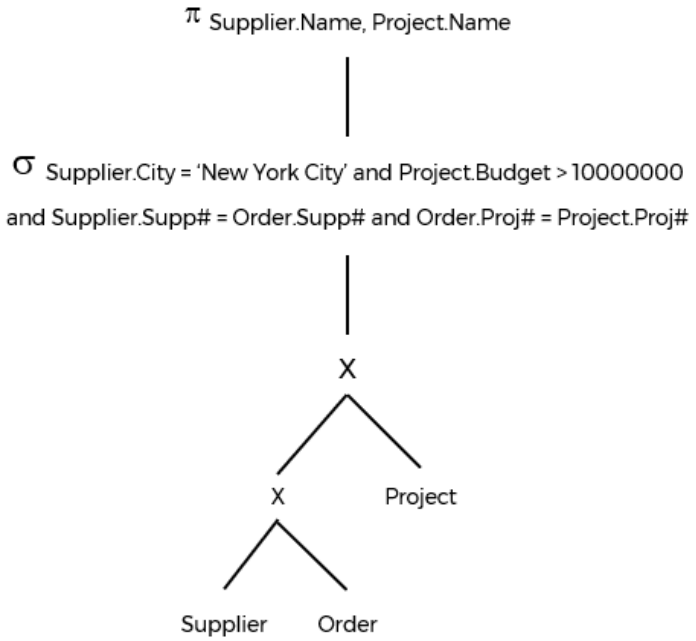
Supplier(Supp#, Name, City, Specialty)
Project(Proj#, Name, City, Budget)
Order(Supp#, Proj#, Part_name, Quantity, Cost)

Apply the four heuristic optimization rules discussed in class to find an efficient execution plan for the following SQL query. It is assumed that there are much more suppliers in New York City than there are projects with budget over 10 million dollars. Draw the corresponding query tree after each rule is applied.

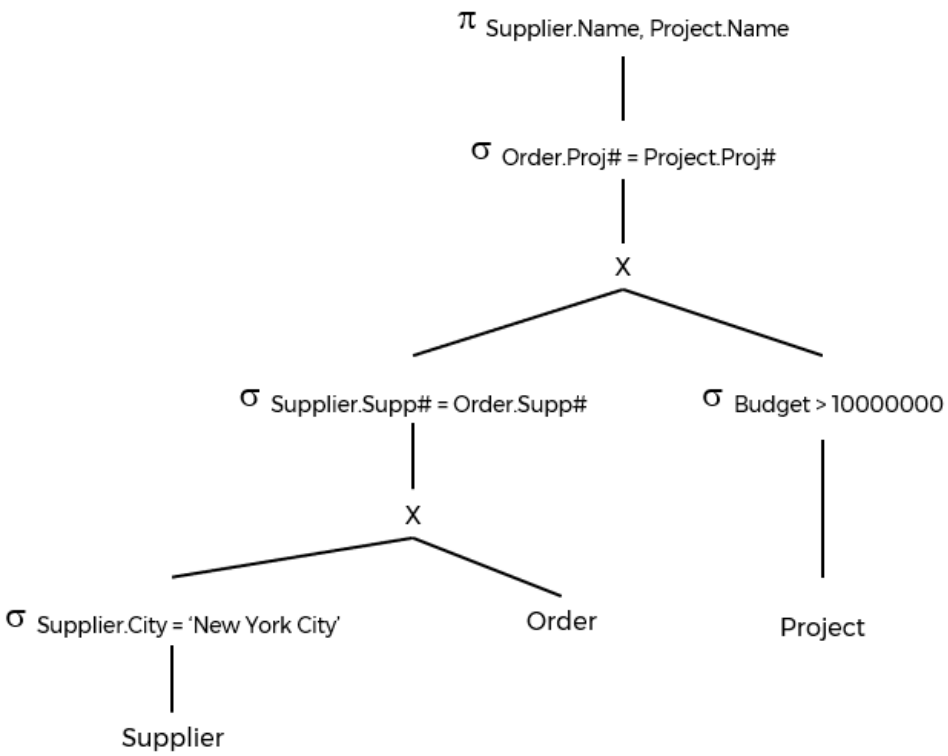
```
select Supplier.Name, Project.Name
from Supplier, Order, Project
where Supplier.City = 'New York City' and Project.Budget > 10000000
and Supplier.Supp# = Order.Supp# and Order.Proj# = Project.Proj#
```

Show the query tree after each optimization rule is applied. Also, write the relational algebra expression corresponding to the fully optimized query tree.

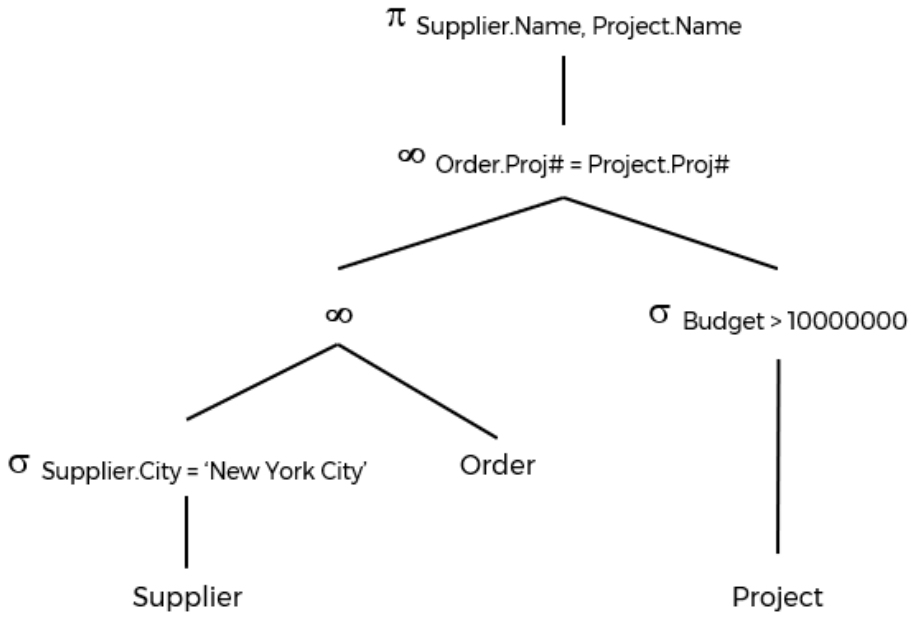
- The given query is -



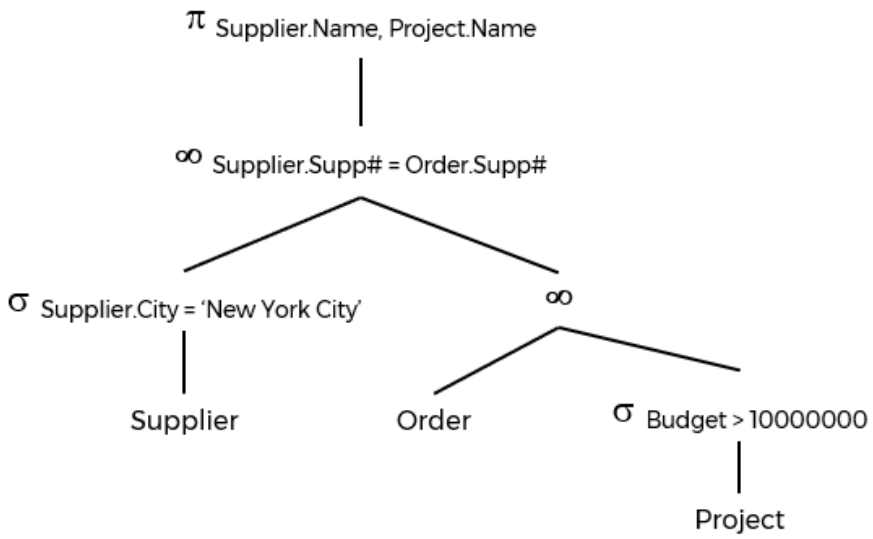
1. Applying rule 1 : Perform selections as early as possible -



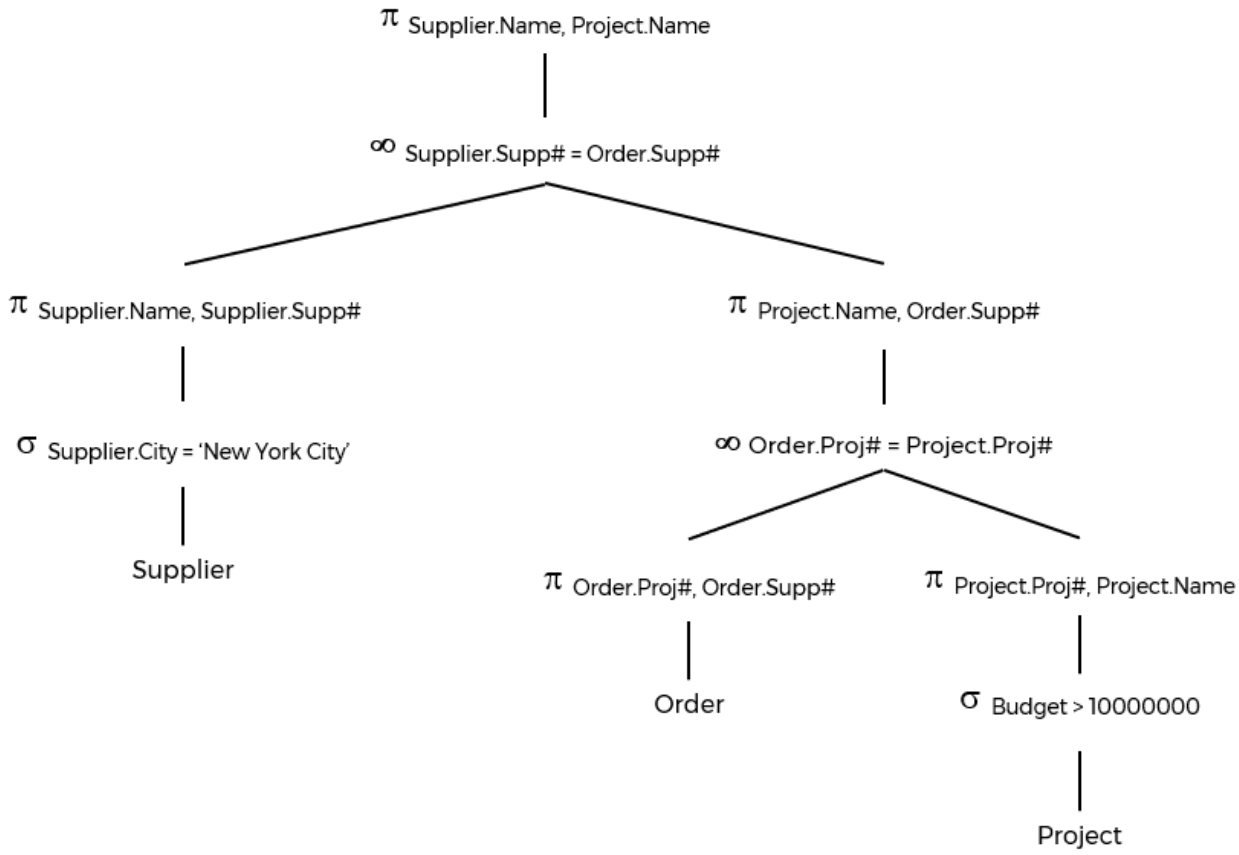
2. Applying rule 2 : Replace Cartesian Products by joins whenever possible -



3. Applying rule 3 : If there are several joins, perform the most restrictive joins first -



4. Applying rule 4 : Project out useless attributes early -



5. The optimized query is -

π Supplier.Name, Project.Name

((π Supplier.Name, Supplie.Supp# (σ Supplier.City = 'New York City' (Supplier)))

∞ Supplier.Supp# = Order.Supp#

(π Project.Name, Order.Supp# (π Order.Proj#, Order.Supp# (Order)

∞ Order.Proj# = Project.Proj#

(π Project.Proj#, Project.Name (σ Budget > 10000000 (Project)))))