CHAPTER 13

Query Optimization

Practice Exercises

13.1 Show that the following equivalences hold. Explain how you can apply them to improve the efficiency of certain queries:

a. $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.

b. $\sigma_\theta({}_A\mathcal{G}_F(E)) = {}_A\mathcal{G}_F(\sigma_\theta(E))$, where θ uses only attributes from A.

c. $\sigma_\theta(E_1 \supset\!\bowtie E_2) = \sigma_\theta(E_1) \supset\!\bowtie E_2$, where θ uses only attributes from E_1.

Answer:

a. $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.
Let us rename $(E_1 \bowtie_\theta (E_2 - E_3))$ as R_1, $(E_1 \bowtie_\theta E_2)$ as R_2 and $(E_1 \bowtie_\theta E_3)$ as R_3. It is clear that if a tuple t belongs to R_1, it will also belong to R_2. If a tuple t belongs to R_3, $t[E_3$'s attributes] will belong to E_3, hence t cannot belong to R_1. From these two we can say that

$$\forall t,\ t \in R_1 \ \Rightarrow\ t \in (R_2 - R_3)$$

It is clear that if a tuple t belongs to $R_2 - R_3$, then $t[R_2$'s attributes] $\in E_2$ and $t[R_2$'s attributes] $\notin E_3$. Therefore:

$$\forall t,\ t \in (R_2 - R_3) \ \Rightarrow\ t \in R_1$$

The above two equations imply the given equivalence.
This equivalence is helpful because evaluation of the right hand side join will produce many tuples which will finally be removed from the result. The left hand side expression can be evaluated more efficiently.

b. $\sigma_\theta({}_A\mathcal{G}_F(E)) = {}_A\mathcal{G}_F(\sigma_\theta(E))$, where θ uses only attributes from A.
θ uses only attributes from A. Therefore if any tuple t in the output of ${}_A\mathcal{G}_F(E)$ is filtered out by the selection of the left hand side, all the tuples in E whose value in A is equal to $t[A]$ are filtered out by the selection of the right hand side. Therefore:

$$\forall t,\ t \notin \sigma_\theta({}_A\mathcal{G}_F(E)) \ \Rightarrow\ t \notin {}_A\mathcal{G}_F(\sigma_\theta(E))$$

Using similar reasoning, we can also conclude that

$$\forall t,\ t \notin {}_A\mathcal{G}_F(\sigma_\theta(E)) \ \Rightarrow\ t \notin \sigma_\theta({}_A\mathcal{G}_F(E))$$

The above two equations imply the given equivalence.
This equivalence is helpful because evaluation of the right hand side avoids performing the aggregation on groups which are anyway going to be removed from the result. Thus the right hand side expression can be evaluated more efficiently than the left hand side expression.

c. $\sigma_\theta(E_1 \supset\!\bowtie E_2) = \sigma_\theta(E_1) \supset\!\bowtie E_2$ where θ uses only attributes from E_1.
θ uses only attributes from E_1. Therefore if any tuple t in the output of $(E_1 \supset\!\bowtie E_2)$ is filtered out by the selection of the left hand side, all the tuples in E_1 whose value is equal to $t[E_1]$ are filtered out by the selection of the right hand side. Therefore:

$$\forall t,\ t \notin \sigma_\theta(E_1 \supset\!\bowtie E_2) \ \Rightarrow\ t \notin \sigma_\theta(E_1) \supset\!\bowtie E_2$$

Using similar reasoning, we can also conclude that

$$\forall t,\ t \notin \sigma_\theta(E_1) \supset\!\bowtie E_2 \ \Rightarrow\ t \notin \sigma_\theta(E_1 \supset\!\bowtie E_2)$$

The above two equations imply the given equivalence.
This equivalence is helpful because evaluation of the right hand side avoids producing many output tuples which are anyway going to be removed from the result. Thus the right hand side expression can be evaluated more efficiently than the left hand side expression.

13.2 For each of the following pairs of expressions, give instances of relations that show the expressions are not equivalent.

a. $\Pi_A(R - S)$ and $\Pi_A(R) - \Pi_A(S)$.

b. $\sigma_{B<4}({}_A\mathcal{G}_{max(B)\ \textbf{as}\ B}(R))$ and ${}_A\mathcal{G}_{max(B)\ \textbf{as}\ B}(\sigma_{B<4}(R))$.

c. In the preceding expressions, if both occurrences of max were replaced by min would the expressions be equivalent?

d. $(R \supset\!\bowtie S) \supset\!\bowtie T$ and $R \supset\!\bowtie (S \supset\!\bowtie T)$
In other words, the natural left outer join is not associative. (Hint: Assume that the schemas of the three relations are $R(a, b1)$, $S(a, b2)$, and $T(a, b3)$, respectively.)

e. $\sigma_\theta(E_1 \supset\!\bowtie E_2)$ and $E_1 \supset\!\bowtie \sigma_\theta(E_2)$, where θ uses only attributes from E_2.

Answer:

a. $R = \{(1, 2)\},\ S = \{(1, 3)\}$
The result of the left hand side expression is $\{(1)\}$, whereas the result of the right hand side expression is empty.

b. $R = \{(1, 2), (1, 5)\}$
The left hand side expression has an empty result, whereas the right hand side one has the result $\{(1, 2)\}$.

c. Yes, on replacing the max by the min, the expressions will become equivalent. Any tuple that the selection in the rhs eliminates would not pass the selection on the lhs if it were the minimum value, and would be eliminated anyway if it were not the minimum value.

d. $R = \{(1, 2)\}$, $S = \{(2, 3)\}$, $T = \{(1, 4)\}$. The left hand expression gives $\{(1, 2, null, 4)\}$ whereas the the right hand expression gives $\{(1, 2, 3, null)\}$.

e. Let R be of the schema (A, B) and S of (A, C). Let $R = \{(1, 2)\}$, $S = \{(2, 3)\}$ and let θ be the expression $C = 1$. The left side expression's result is empty, whereas the right side expression results in $\{(1, 2, null)\}$.

13.3 SQL allows relations with duplicates (Chapter 3).

a. Define versions of the basic relational-algebra operations σ, Π, $\times$, $\bowtie$, $-$, $\cup$, and $\cap$ that work on relations with duplicates, in a way consistent with SQL.

b. Check which of the equivalence rules 1 through 7.b hold for the multiset version of the relational-algebra defined in part a.

Answer:

a. We define the multiset versions of the relational-algebra operators here. Given multiset relations r_1 and r_2,

i. σ
Let there be c_1 copies of tuple t_1 in r_1. If t_1 satisfies the selection σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$, otherwise there are none.

ii. Π
For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$, where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.

iii. $\times$
If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, then there are $c_1 * c_2$ copies of the tuple $t_1.t_2$ in $r_1 \times r_2$.

iv. $\bowtie$
The output will be the same as a cross product followed by a selection.

v. $-$
If there are c_1 copies of tuple t in r_1 and c_2 copies of t in r_2, then there will be $c_1 - c_2$ copies of t in $r_1 - r_2$, provided that $c_1 - c_2$ is positive.

vi. $\cup$

If there are c_1 copies of tuple t in r_1 and c_2 copies of t in r_2, then there will be $c_1 + c_2$ copies of t in $r_1 \cup r_2$.

vii. $\cap$
If there are c_1 copies of tuple t in r_1 and c_2 copies of t in r_2, then there will be $min(c_1, c_2)$ copies of t in $r_1 \cap r_2$.

b. All the equivalence rules 1 through 7.b of section 13.2.1 hold for the multiset version of the relational-algebra defined in the first part. There exist equivalence rules which hold for the ordinary relational-algebra, but do not hold for the multiset version. For example consider the rule :-

$$A \cap B \quad = \quad A \cup B \; - \; (A - B) \; - \; (B - A)$$

This is clearly valid in plain relational-algebra. Consider a multiset instance in which a tuple t occurs 4 times in A and 3 times in B. t will occur 3 times in the output of the left hand side expression, but 6 times in the output of the right hand side expression. The reason for this rule to not hold in the multiset version is the asymmetry in the semantics of multiset union and intersection.

13.4 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$, with primary keys A, C, and E, respectively. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples, and r_3 has 750 tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$, and give an efficient strategy for computing the join.

Answer:

- The relation resulting from the join of r_1, r_2, and r_3 will be the same no matter which way we join them, due to the associative and commutative properties of joins. So we will consider the size based on the strategy of $((r_1 \bowtie r_2) \bowtie r_3)$. Joining r_1 with r_2 will yield a relation of at most 1000 tuples, since C is a key for r_2. Likewise, joining that result with r_3 will yield a relation of at most 1000 tuples because E is a key for r_3. Therefore the final relation will have at most 1000 tuples.
- An efficient strategy for computing this join would be to create an index on attribute C for relation r_2 and on E for r_3. Then for each tuple in r_1, we do the following:
  a. Use the index for r_2 to look up at most one tuple which matches the C value of r_1.
  b. Use the created index on E to look up in r_3 at most one tuple which matches the unique value for E in r_2.

13.5 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$ of Practice Exercise 13.4. Assume that there are no primary keys, except the entire schema. Let $V(C, r_1)$ be 900, $V(C, r_2)$ be 1100, $V(E, r_2)$ be 50, and $V(E, r_3)$ be 100. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples, and r_3 has 750

tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$ and give an efficient strategy for computing the join.

Answer: The estimated size of the relation can be determined by calculating the average number of tuples which would be joined with each tuple of the second relation. In this case, for each tuple in r_1, $1500/V(C, r_2)$ = 15/11 tuples (on the average) of r_2 would join with it. The intermediate relation would have 15000/11 tuples. This relation is joined with r_3 to yield a result of approximately 10,227 tuples ($15000/11 \times 750/100 = 10227$). A good strategy should join r_1 and r_2 first, since the intermediate relation is about the same size as r_1 or r_2. Then r_3 is joined to this result.

13.6 Suppose that a B^+-tree index on *building* is available on relation *department*, and that no other index is available. What would be the best way to handle the following selections that involve negation?

a. $\sigma_{\neg(building < \text{"Watson"})}(department)$

b. $\sigma_{\neg(building = \text{"Watson"})}(department)$

c. $\sigma_{\neg(building < \text{"Watson"} \vee budget < 50000)}(department)$

Answer:

a. Use the index to locate the first tuple whose *building* field has value "Watson". From this tuple, follow the pointer chains till the end, retrieving all the tuples.

b. For this query, the index serves no purpose. We can scan the file sequentially and select all tuples whose *building* field is anything other than "Watson".

c. This query is equivalent to the query:

$$\sigma_{building \geq \text{'Watson'} \wedge budget < 5000)}(department).$$

Using the *building* index, we can retrieve all tuples with *building* value greater than or equal to "Watson" by following the pointer chains from the first "Watson" tuple. We also apply the additional criteria of $budget < 5000$ on every tuple.

13.7 Consider the query:

```
select *
from r, s
where upper(r.A) = upper(s.A);
```

where "upper" is a function that returns its input argument with all lowercase letters replaced by the corresponding uppercase letters.

a. Find out what plan is generated for this query on the database system you use.

b. Some database systems would use a (block) nested-loop join for this query, which can be very inefficient. Briefly explain how hash-join or merge-join can be used for this query.

Answer:

a. First create relations r and s, and add some tuples to the two relations, before finding the plan chosen; or use existing relations in place of r and s. Compare the chosen plan with the plan chosen for a query directly equating $r.A = s.B$. Check the estimated statistics too. Some databases may give the same plan, but with vastly different statistics.
(On PostgreSQL, we found that the optimizer used the merge join plan described in the answer to the next part of this question.)

b. To use hash join, hashing should be done after applying the upper() function to $r.A$ and $s.A$. Similarly, for merge join, the relations should be sorted on the result of applying the upper() function on $r.A$ and $s.A$. The hash or merge join algorithms can then be used unchanged.

13.8 Give conditions under which the following expressions are equivalent

$$_{A,B}\mathcal{G}_{agg(C)}(E_1 \bowtie E_2) \quad \text{and} \quad (_{A}\mathcal{G}_{agg(C)}(E_1)) \bowtie E_2$$

where *agg* denotes any aggregation operation. How can the above conditions be relaxed if *agg* is one of **min** or **max**?
Answer: The above expressions are equivalent provided E_2 contains only attributes A and B, with A as the primary key (so there are no duplicates). It is OK if E_2 does not contain some A values that exist in the result of E_1, since such values will get filtered out in either expression. However, if there are duplicate values in $E_2.A$, the aggregate results in the two cases would be different.
If the aggregate function is min or max, duplicate A values do not have any effect. However, there should be no duplicates on (A, B); the first expression removes such duplicates, while the second does not.

13.9 Consider the issue of interesting orders in optimization. Suppose you are given a query that computes the natural join of a set of relations S. Given a subset $S1$ of S, what are the interesting orders of $S1$?
Answer: The interesting orders are all orders on subsets of attributes that can potentially participate in join conditions in further joins. Thus, let T be the set of all attributes of $S1$ that also occur in any relation in $S - S1$. Then every ordering of every subset of T is an interesting order.

13.10 Show that, with n relations, there are $(2(n-1))!/(n-1)!$ different join orders. *Hint:* A **complete binary tree** is one where every internal node has exactly two children. Use the fact that the number of different complete

binary trees with n leaf nodes is:

$$\frac{1}{n}\binom{2(n-1)}{(n-1)}$$

If you wish, you can derive the formula for the number of complete binary trees with n nodes from the formula for the number of binary trees with n nodes. The number of binary trees with n nodes is:

$$\frac{1}{n+1}\binom{2n}{n}$$

This number is known as the **Catalan number**, and its derivation can be found in any standard textbook on data structures or algorithms.

Answer: Each join order is a complete binary tree (every non-leaf node has exactly two children) with the relations as the leaves. The number of different complete binary trees with n leaf nodes is $\frac{1}{n}\binom{2(n-1)}{(n-1)}$. This is because there is a bijection between the number of complete binary trees with n leaves and number of binary trees with $n-1$ nodes. Any complete binary tree with n leaves has $n-1$ internal nodes. Removing all the leaf nodes, we get a binary tree with $n-1$ nodes. Conversely, given any binary tree with $n-1$ nodes, it can be converted to a complete binary tree by adding n leaves in a unique way. The number of binary trees with $n-1$ nodes is given by $\frac{1}{n}\binom{2(n-1)}{(n-1)}$, known as the Catalan number. Multiplying this by $n!$ for the number of permutations of the n leaves, we get the desired result.

13.11 Show that the lowest-cost join order can be computed in time $O(3^n)$. Assume that you can store and look up information about a set of relations (such as the optimal join order for the set, and the cost of that join order) in constant time. (If you find this exercise difficult, at least show the looser time bound of $O(2^{2n})$.)

Answer: Consider the dynamic programming algorithm given in Section 13.4. For each subset having $k+1$ relations, the optimal join order can be computed in time 2^{k+1}. That is because for one particular pair of subsets A and B, we need constant time and there are at most $2^{k+1}-2$ different subsets that A can denote. Thus, over all the $\binom{n}{k+1}$ subsets of size $k+1$, this cost is $\binom{n}{k+1}2^{k+1}$. Summing over all k from 1 to $n-1$ gives the binomial expansion of $((1+x)^n - x)$ with $x=2$. Thus the total cost is less than 3^n.

13.12 Show that, if only left-deep join trees are considered, as in the System R optimizer, the time taken to find the most efficient join order is around $n2^n$. Assume that there is only one interesting sort order.

Answer: The derivation of time taken is similar to the general case, except that instead of considering $2^{k+1}-2$ subsets of size less than or equal to

k for A, we only need to consider $k+1$ subsets of size exactly equal to k. That is because the right hand operand of the topmost join has to be a single relation. Therefore the total cost for finding the best join order for all subsets of size $k+1$ is $\binom{n}{k+1}(k+1)$, which is equal to $n\binom{n-1}{k}$. Summing over all k from 1 to $n-1$ using the binomial expansion of $(1+x)^{n-1}$ with $x=1$, gives a total cost of less than $n2^{n-1}$.

13.13 Consider the bank database of Figure 13.9, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Write a nested query on the relation *account* to find, for each branch with name starting with B, all accounts with the maximum balance at the branch.

b. Rewrite the preceding query, without using a nested subquery; in other words, decorrelate the query.

c. Give a procedure (similar to that described in Section 13.4.4) for decorrelating such queries.

Answer:

a. The nested query is as follows:

```
select  S.acount_number
from    account S
where   S.branch_name like 'B%' and
        S.balance =
        (select max(T.balance)
        from account T
        where T.branch_name = S.branch_name)
```

b. The decorrelated query is as follows:

```
create table t1 as
        select branch_name, max(balance)
        from account
        group by branch_name
select  account_number
from    account, t1
where   account.branch_name like 'B%' and
        account.branch_name = t1.branch_name and
        account.balance = t1.balance
```

c. In general, consider the queries of the form:

select $\cdots$
from L_1
where P_1 **and**
A_1 **op**
(**select f**(A_2)
from L_2
where P_2)

where, f is some aggregate function on attributes A_2, and *op* is some boolean binary operator. It can be rewritten as

create table t_1 **as**
select f(A_2),V
from L_2
where P_2^1
group by V
select $\cdots$
from L_1, t_1
where P_1 **and** P_2^2 **and**
A_1 **op** $t_1.A_2$

where P_2^1 contains predicates in P_2 without selections involving correlation variables, and P_2^2 introduces the selections involving the correlation variables. V contains all the attributes that are used in the selections involving correlation variables in the nested query.

13.14 The set version of the semijoin operator $\ltimes$ is defined as follows:

$$r \ltimes_\theta s = \Pi_R(r \bowtie_\theta s)$$

where R is the set of attributes in the schema of r. The multiset version of the semijoin operation returns the same set of tuples, but each tuple has exactly as many copies as it had in r.
Consider the nested query we saw in Section 13.4.4 which finds the names of all instructors who taught a course in 2007. Write the query in relational algebra using the multiset semjoin operation, ensuring that the number of duplicates of each name is the same as in the SQL query. (The semijoin operation is widely used for decorrelation of nested queries.)
Answer: The query can be written as follows:
$instructor \ltimes_{instructor.ID=teaches.ID} (\sigma_{year=2007}(teaches))$