

CHAPTER 16



Query Optimization

Practice Exercises

- 16.1** Download the university database schema and the large university dataset from dbbook.com. Create the university schema on your favorite database, and load the large university dataset. Use the **explain** feature described in Note 16.1 on page 746 to view the plan chosen by the database, in different cases as detailed below.
- Write a query with an equality condition on *student.name* (which does not have an index), and view the plan chosen.
 - Create an index on the attribute *student.name*, and view the plan chosen for the above query.
 - Create simple queries joining two relations, or three relations, and view the plans chosen.
 - Create a query that computes an aggregate with grouping, and view the plan chosen.
 - Create an SQL query whose chosen plan uses a semijoin operation.
 - Create an SQL query that uses a **not in** clause, with a subquery using aggregation. Observe what plan is chosen.
 - Create a query for which the chosen plan uses correlated evaluation (the way correlated evaluation is represented varies by database, but most databases would show a filter or a project operator with a subplan or subquery).
 - Create an SQL update query that updates a single row in a relation. View the plan chosen for the update query.

- i. Create an SQL update query that updates a large number of rows in a relation, using a subquery to compute the new value. View the plan chosen for the update query.

Answer:

The answer depends on the database.

FILL IN Suggested queries for each exercise as verified on some database

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

- a. $E_1 \bowtie_{\theta} (E_2 - E_3) \equiv (E_1 \bowtie_{\theta} E_2 - E_1 \bowtie_{\theta} E_3)$.
- b. $\sigma_{\theta}(A\gamma_F(E)) \equiv A\gamma_F(\sigma_{\theta}(E))$, where θ uses only attributes from A .
- c. $\sigma_{\theta}(E_1 \bowtie E_2) \equiv \sigma_{\theta}(E_1) \bowtie E_2$, 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. $\sigma_{\theta}(A\gamma_F(E)) = A\gamma_F(\sigma_{\theta}(E))$, where θ uses only attributes from A .

θ uses only attributes from A . Therefore if any tuple t in the output of $A\gamma_F(E)$ 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:

$$\forall t, t \notin \sigma_{\theta}(A\gamma_F(E)) \Rightarrow t \notin A\gamma_F(\sigma_{\theta}(E))$$

Using similar reasoning, we can also conclude that

$$\forall t, t \notin A\gamma_F(\sigma_{\theta}(E)) \Rightarrow t \notin \sigma_{\theta}(A\gamma_F(E))$$

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

θ uses only attributes from E_1 . Therefore if any tuple t in the output of $(E_1 \bowtie E_2)$ 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:

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

Using similar reasoning, we can also conclude that

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

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 going to be removed from the result. Thus the right-hand side expression can be evaluated more efficiently than the left-hand side expression.

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

- $\Pi_A(r - s)$ and $\Pi_A(r) - \Pi_A(s)$.
- $\sigma_{B < 4}(\text{AGG}_{\max(B)} \text{ as } B(r))$ and $\text{AGG}_{\max(B)} \text{ as } B(\sigma_{B < 4}(r))$.
- In the preceding expressions, if both occurrences of *max* were replaced by *min*, would the expressions be equivalent?
- $(r \bowtie s) \bowtie t$ and $r \bowtie (s \bowtie t)$
In other words, the natural right outer join is not associative.
- $\sigma_\theta(E_1 \bowtie E_2)$ and $E_1 \bowtie \sigma_\theta(E_2)$, where θ uses only attributes from E_2 .

Answer:

- $R = \{(1, 2)\}$, $S = \{(1, 3)\}$
The result of the left-hand side expression is $\{(1)\}$, whereas the result of the right-hand side expression is empty.
- $R = \{(1, 2), (1, 5)\}$
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 $\{(1, 2, \text{null}, 4)\}$ whereas the right-hand expression gives $\{(1, 2, 3, \text{null})\}$.
- 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 $\{(1, 2, \text{null})\}$.

16.4 SQL allows relations with duplicates (Chapter 3), and the multiset version of the relational algebra is defined in Note 3.1 on page 80, Note 3.2 on page 97, and Note 3.3 on page 108. Check which of the equivalence rules 1 through 7.b hold for the multiset version of the relational algebra.

Answer:

All the equivalence rules 1 through 7.b of section Section 16.2.1 hold for the multiset version of the relational algebra defined in Chapter 2.

There exist equivalence rules that hold for the ordinary relational algebra but do not hold for the multiset version. For example consider the rule :-

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

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.

16.5 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 .

16.6 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$ of Practice Exercise 16.5. 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.

- 16.7** 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. $\sigma_{\neg(\text{building} < \text{"Watson"})}(\text{department})$
 - b. $\sigma_{\neg(\text{building} = \text{"Watson"})}(\text{department})$
 - c. $\sigma_{\neg(\text{building} < \text{"Watson"} \vee \text{budget} < 50000)}(\text{department})$

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:

$$\sigma_{\text{building} \geq \text{'Watson'} \wedge \text{budget} < 50000}(\text{department}).$$

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.

16.8 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.

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

- 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.)

- 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.

16.9 Give conditions under which the following expressions are equivalent:

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

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.

- 16.10** 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 S_1 of S , what are the interesting orders of S_1 ?

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 S_1 that also occur in any relation in $S - S_1$. Then every ordering of every subset of T is an interesting order.

- 16.11** Modify the $\text{FindBestPlan}(S)$ function to create a function $\text{FindBestPlan}(S, O)$, where O is a desired sort order for S , and which considers interesting sort orders. A *null* order indicates that the order is not relevant. *Hints:* An algorithm A may give the desired order O ; if not a sort operation may need to be added to get the desired order. If A is a merge-join, FindBestPlan must be invoked on the two inputs with the desired orders for the inputs.

Answer:

FILL IN

- 16.12** 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.

- 16.13** 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 $O(2^{2^n})$.)

Answer:

Consider the dynamic programming algorithm given in Section 16.4. 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 .

- 16.14** 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 $n2^{n-1}$.

- 16.15** Consider the bank database of Figure 16.9, where the primary keys are underlined. Construct the following SQL queries for this relational database.
- 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.
 - Rewrite the preceding query without using a nested subquery; in other words, decorrelate the query, but in SQL.
 - Give a relational algebra expression using semijoin equivalent to the query.

- d. Give a procedure (similar to that described in Section 16.4.4) for decorrelating such queries.

Answer:

- a. The nested query is as follows:

```

select    S.account_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. FILL IN
- d. In general, consider the queries of the form:

```

branch(branch_name, branch_city, assets)
customer(customer_name, customer_street, customer_city)
loan(loan_number, branch_name, amount)
borrower(customer_name, loan_number)
account(account_number, branch_name, balance)
depositor(customer_name, account_number)

```

Figure 16.9 Banking database.

```

select  ...
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

***** FILL IN ***** GIVE Relational algebra version *****

```

create table  $t_1$  as
    select f( $A_2$ ),  $V$ 
    from  $L_2$ 
    where  $P_2^1$ 
    group by  $V$ 
select  ...
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.