JOURNAL OF ENGINEERING
annals OF
FACULTY OF ENGINEERING HUNEDOARA
Tome V (year 2007), Fascicole 1, (ISSN 1584 – 2665)

AVOID REDUNDANT JOIN CONDITION IN SQL QUERIES

PETRINI Mircea

UNIVERSITY OF PETROŞANI

ABSTRACT:
This paper presents a technique to avoid redundant join condition in SQL queries by forcing execution order for outer queries and subqueries. We convert between an EXISTS condition on a correlated subquery and the equivalent IN condition on a noncorrelated subquery.
KEYWORDS:
SQL, redundant, join, queries

1. INTRODUCTION IN REDUNDANT JOIN CONDITIONS

Normally, between any number of tables, the join count is the number of tables minus one. For example, between three tables, we expect to find two joins. Occasionally, a query permits an extra, redundant join. For example, if we have an Addresses table that contains all addresses significant to the company, it might have a one-to-zero or one-to-one relationship with the earlier Locations table, which contains only locations owned by the company and which references Addresses through a matching primary key. In this case, we might find a query like the following:

```
SELECT ...
FROM Employees E, Locations L, Addresses A
WHERE E.Location_ID=L.Location_ID
   AND E.Location_ID=A.Address_ID
   AND A.ZIP_Code=95628
```

By transitivity (if a=b and b=c, then a=c), we can deduce that the condition L.Location_ID=A.Address_ID must be true for all rows this query would return. However, that condition is not explicit in the query, and not all databases will deduce it and fill it in if it is left out. The best plan, in this case, will likely begin with all addresses within that ZIP Code and immediately join to Locations to discard all addresses except the one or two that correspond to company locations, before joining to Employees. Since that join order requires the missing join condition to support an indexed path from Addresses to Locations, we should make the missing join condition explicit:

```
SELECT ...
FROM Employees E, Locations L, Addresses A
WHERE E.Location_ID=L.Location_ID
   AND E.Location_ID=A.Address_ID
   AND L.Location_ID=A.Address_ID
   AND A.ZIP_Code=95628
```

Since we do not want to follow the join from Addresses to Employees directly, we could also remove, if necessary, the redundant join condition E.Location_ID=A.Address_ID, to discourage that unwanted join operation.

2. AVOID UNUSUALLY JOIN ORDERS

Forcing joins in the direction we want, using the earlier techniques for preventing use of the wrong indexes, will prevent many undesired join orders. What do we do when we want the database to follow a particular join direction eventually, but not too early in the execution plan? We cannot afford to disable an index, because we must use that index eventually, just not too early. Consider the following two joins, in which we want to start the query with reads of T1 and then join to T2 before joining to T3:

```
... AND T1.Key2_ID=T2.Key2_ID
AND T1.Key3_ID=T3.Key3_ID ...
```

Here, we want to follow nested loops into both T2 and T3, following indexes in the keys mentioned and reaching T2 before reaching T3. To postpone the join we want to happen later, make it depend (or at least to appear to depend) on data from the join that must happen earlier. Here is a solution:

```
... AND T1.Key2_ID=T2.Key2_ID
AND T1.Key3_ID+0*T2.Key2_ID=T3.Key3_ID ...
```

We and I know that the second version is logically equivalent to the first. However, the database just finds an expression on the left side of the second join that depends on both T1 and T2 (not recognizing that no value from T2 can change the result), so it won't try to perform the join to T3 until after T2.

If necessary, we can string together joins like this to completely constrain a join order. For each join after the first, add a logically irrelevant component referencing one of the columns added in the preceding join to the join expression. For example, if we want to reach tables T1 through T5 in numerical order, we can use the following. Notice that the join condition for the T3 table uses the expression 0*T2.Key2_ID to force the join to T2 to occur first. Likewise, the join condition for the T4 table uses 0*T3.Key3_ID to force T3 to be joined first.

```
... AND T1.Key2_ID=T2.Key2_ID
AND T1.Key3_ID+0*T2.Key2_ID=T3.Key3_ID
AND T1.Key4_ID+0*T3.Key3_ID=T4.Key4_ID
AND T1.Key4_ID+0*T4.Key4_ID=T5.Key5_ID ...
```

I'll apply this method to a concrete example. Consider the following SQL:

```
SELECT E.First_Name, E.Last_Name, E.Salary, LE.Description,
      M.First_Name, M.Last_Name, LM.Description
FROM Locations LE, Locations LM, Employees M, Employees E
WHERE E.Last_Name = 'Johnson'
  AND E.Manager_ID=M.Employee_ID
  AND E.Location_ID=LE.Location_ID
  AND M.Location_ID=LM.Location_ID
  AND LE.Description='Dallas'
```

Assume that we have an execution plan that drives from the index on the employee's last name, but we find that the join to the employee's location (alias LE) to discard employees at locations other than Dallas is unfortunately happening last, after the other joins (to M and LM). We should join to LE immediately from E, to minimize the number of rows we need to join to the other two tables. Starting from E, the join to LM is not immediately possible, so if we prevent the join to M before LE, we should get the join order we want. Here's how:

```
SELECT E.First_Name, E.Last_Name, E.Salary, LE.Description,
      M.First_Name, M.Last_Name, LM.Description
FROM Locations LE, Locations LM, Employees M, Employees E
WHERE E.Last_Name = 'Johnson'
  AND E.Manager_ID+0*LE.Location_ID=M.Employee_ID
  AND E.Location_ID=LE.Location_ID
  AND M.Location_ID=LM.Location_ID
  AND LE.Description='Dallas'
```

The key here is that I've made the join to M dependent on the value from LE. The expression 0*LE.Location_ID forces the optimizer to join to LE before M. Because of the multiply-by-zero, the added expression has no effect on the results returned by the query.

3. FORCING EXECUTION ORDER FOR OUTER QUERIES AND SUBQUERIES

In the next example, we can convert this:

```
SELECT ...
FROM Departments D
WHERE EXISTS (SELECT NULL FROM Employees E
              WHERE E.Department_ID=D.Department_ID)
```

to this:

```
SELECT ...
FROM Departments D
WHERE D.Department_ID IN (SELECT E.Department_ID FROM Employees E)
```

The first form implies that the database drives from the outer query to the subquery. For each row returned by the outer query, the database executes the join in the subquery. The second form implies that we begin with the list of distinct departments that have employees, as found in the noncorrelated subquery, and drive from that list into the matching list of such departments in the outer query. Sometimes, the database itself follows this implied join order, although some databases can make the conversion internally if their optimizer finds that the alternate order is better. To make your own SQL more readable and to make it work well regardless of whether your database can convert the forms internally, use the form that implies the order we want. To force that order even when the database could make the conversion, use the same join-direction-forcing technique. Thus, an EXISTS condition that forces the outer query to execute first would look like this:

```
SELECT ...
FROM Departments D
WHERE EXISTS (SELECT NULL FROM Employees E
              WHERE E.Department_ID=D.Department_ID+0)
```

For the contrary order, an IN condition that forces the implied driving order from the subquery to the outer query would look like this:

```
SELECT ...
FROM Departments D
WHERE D.Department_ID IN (SELECT E.Department_ID+0 FROM Employees E)
```

We can have several subqueries in which the database either must drive from the outer query to the subquery (such as NOT EXISTS subqueries) or should drive in that order. Such a case implies a choice of the order of execution of the subqueries. We can also have choices about whether to execute subqueries after completing the outer query, or at the first opportunity, as soon as the correlation join is possible, or at some point between these extremes.

The first tactic for controlling the order of subquery execution is simply to list the subqueries in order in the WHERE clause (i.e., the top subquery to be executed should be listed first). This is one of the few times when WHERE-clause order seems to matter.

Rarely, the database will execute a subquery sooner than we would like. The same tactic for postponing joins works for correlation joins, the joins in subqueries that correlate the subqueries to the outer queries. For example, consider this query:

```
SELECT ...
FROM Orders O, Customers C, Regions R
WHERE O.Status_Code='OP'
  AND O.Customer_ID=C.Customer_ID
  AND C.Customer_Type_Code='GOV'
  AND C.Region_ID=R.Region_ID
  AND EXISTS (SELECT NULL
        FROM Order_Details OD
```

```
          WHERE O.Order_ID=OD.Order_ID
            AND OD.Shipped_Flag='Y')
```

For this query we might find that the subquery runs as soon as we reach the driving Orders table, but we might wish to perform the join to Customers first, to discard nongovernmental orders, before we take the expense of the subquery execution. In this case, this would be the transformation to postpone the correlation join:

```
SELECT ...
FROM Orders O, Customers C, Regions R
WHERE O.Status_Code='OP'
  AND O.Customer_ID=C.Customer_ID
  AND C.Customer_Type_Code='GOV'
  AND C.Region_ID=R.Region_ID
  AND EXISTS (SELECT NULL
          FROM Order_Details OD
          WHERE O.Order_ID+0*C.Customer_ID=OD.Order_ID
            AND OD.Shipped_Flag='Y')
```

Notice the addition of +0*C.Customer_ID to the subquery's WHERE clause. This ensures the join to Customers occurs first, before the subquery executes.

4. CONCLUSIONS

Most queries with subqueries can logically drive from either the outer query or the subquery. Depending on the selectivity of the subquery condition, either choice can be best. The choice generally arises for queries with EXISTS or IN conditions. We can always convert between an EXISTS condition on a correlated subquery and the equivalent IN condition on a noncorrelated subquery.

BIBLIOGRAPHY

[1.] FOTACHE M., STRÎMBEI C., CRETU L – ORACLE 9i2 - Ghidul dezvoltării aplicațiilor profesionale, Ed., Teora, 2005

[2.] FOTACHE M. – Dialecte SQL, Ed. Gh. Asachi, 2002

[3.] HENDERSON K. – Transact SQL, Ed. Teora, 2003

[4.] HENDERSON K. – Guru's Guide to Transact SQL, Ed. Addison-Wesley, 2000

[5.] Oracle Co – SQL*Plus User's Guide and Reference, 2003