Microsoft

T-SQL Querying

Professional

SolidQ

Itzik Ben-Gan
Dejan Sarka
Adam Machanic
Kevin Farlee

Microsoft

T-SQL Querying

Itzik Ben-Gan
Dejan Sarka
Adam Machanic
Kevin Farlee

```
  LEFT OUTER JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
  LEFT OUTER JOIN Production.Products AS P
    ON P.productid = OD.productid
  LEFT OUTER JOIN Production.Suppliers AS S
    ON S.supplierid = P.supplierid;
```

There are two problems with this solution. One problem is that you actually changed the meaning of the query from the intended one. You were after a left outer join between Customers and the result of the inner joins between the remaining tables. Rows without matches in the remaining tables aren't supposed to be preserved; however, with this solution, if they are left rows, they will be preserved. The other problem is that the optimizer has less flexibility in altering the join order when dealing with outer joins. Pretty much the only flexibility that it does have is to change T1 LEFT OUTER JOIN T2 to T2 RIGHT OUTER JOIN T1.

A better solution is to start with the inner joins between the tables besides Customers and then to apply a right outer join with Customers, like so:

```
SELECT DISTINCT C.companyname AS customer, S.companyname AS supplier
FROM Sales.Orders AS O
  INNER JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
  INNER JOIN Production.Products AS P
    ON P.productid = OD.productid
  INNER JOIN Production.Suppliers AS S
    ON S.supplierid = P.supplierid
  RIGHT OUTER JOIN Sales.Customers AS C
    ON C.custid = O.custid;
```

Now the solution is both semantically correct and leaves more flexibility to the optimizer to apply join-ordering optimization. This query returns 1,238 rows, which include the two customers who didn't place orders.

There's an even more elegant solution that allows you to express the joins in a manner similar to the way you think of the task. Remember you want to apply a left outer join between Customers and a unit representing the result of the inner joins between the remaining tables. To define such a unit, simply enclose the inner joins in parentheses and move the join predicate that relates Customers and that unit after the parentheses, like so:

```
SELECT DISTINCT C.companyname AS customer, S.companyname AS supplier
FROM Sales.Customers AS C
  LEFT OUTER JOIN
      (     Sales.Orders AS O
       INNER JOIN Sales.OrderDetails AS OD
         ON OD.orderid = O.orderid
       INNER JOIN Production.Products AS P
         ON P.productid = OD.productid
       INNER JOIN Production.Suppliers AS S
         ON S.supplierid = P.supplierid)
    ON O.custid = C.custid;
```

Curiously, the parentheses are not really required. If you remove them and run the code, you will see it runs successfully and returns the correct result. What really makes the difference is the parentheses-like arrangement of the units and the placement of the ON clauses reflecting this arrangement. Simply make sure that the ON clause that is supposed to relate two units appears right after them; otherwise, the query will not be a valid one. With this in mind, following is another valid arrangement of the units, achieving the desired result for our task:

```
SELECT DISTINCT C.companyname AS customer, S.companyname AS supplier
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
  INNER JOIN Sales.OrderDetails AS OD
  INNER JOIN Production.Products AS P
  INNER JOIN Production.Suppliers AS S
    ON S.supplierid = P.supplierid
    ON P.productid = OD.productid
    ON OD.orderid = O.orderid
    ON O.custid = C.custid;
```

I find that using parentheses accompanied with adequate indentation results in much clearer code, so I recommend using them even though they are not required.

Earlier I mentioned that one of the heuristics the optimizer uses to reduce optimization time is to not consider bushy plans that involve joins between results of joins. If you suspect that such a bushy plan is the most efficient and want the optimizer to use it, you need to force it. The way you achieve this is by defining two units, each based on a join with the respective ON clause, followed by an ON clause that relates the two units. You also need to specify the FORCE ORDER query option.

Following is an example of forcing such a bushy plan:

```
SELECT DISTINCT C.companyname AS customer, S.companyname AS supplier
FROM Sales.Customers AS C
  INNER JOIN
          (Sales.Orders AS O INNER JOIN Sales.OrderDetails AS OD
             ON OD.orderid = O.orderid)
      INNER JOIN
          (Production.Products AS P INNER JOIN Production.Suppliers AS S
             ON S.supplierid = P.supplierid)
        ON P.productid = OD.productid
    ON O.custid = C.custid
OPTION (FORCE ORDER);
```

Here you have one unit (call it *U1*) based on a join between Orders and OrderDetails. You have another unit (call it *U2*) based on a join between Products and Suppliers. Then you have a join between the two units (call the result *U1-2*). Then you have a join between Customers and U1-2. The execution plan for this query is shown in Figure 3-18.

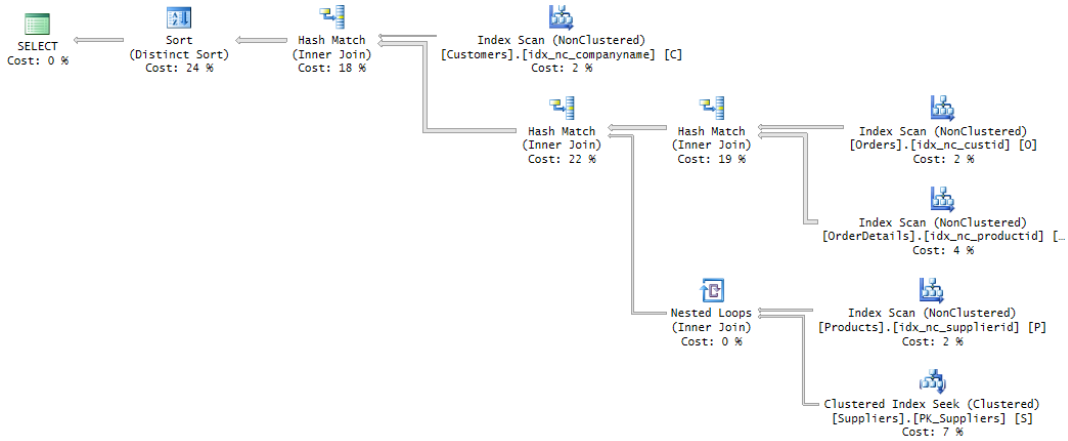

FIGURE 3-18 Bushy plan.

Observe that the desired bushy layout was achieved in the plan.

Semi and anti semi joins

Normally, a join matches rows from two tables and returns elements from both sides. What makes a join a *semi join* is that you return elements from only one of the sides. The side you return the elements from determines whether it's a left or right semi join.

As an example, consider a request to return the customer ID and company name of customers who placed orders. You return information only from the Customers table, provided that a matching row is found in the Orders table. Considering the Customers table as the left table, the operation is a left semi join. There are a number of ways to implement the task. One is to use an inner join and apply a DISTINCT clause to remove duplicate customer info, like so:

```
SELECT DISTINCT C.custid, C.companyname
FROM Sales.Customers AS C
  INNER JOIN Sales.Orders AS O
    ON O.custid = C.custid;
```

Another is to use the EXISTS predicate, like so:

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE EXISTS(SELECT *
             FROM Sales.Orders AS O
             WHERE O.custid = C.custid);
```

Both queries get the same plan. Of course, there are more ways to achieve the task.

An anti semi join is one in which you return elements from one table if a matching row cannot be found in a related table. Also here, depending on whether you return information from the left table

or the right one, the join is either a left or right anti semi join. As an example, a request to return customers who did not place orders is fulfilled with a left anti semi join operation.

There are a number of classic ways to achieve such an anti semi join. One is to use a left outer join between Customers and Orders and filter only outer rows, like so:

```
SELECT C.custid, C.companyname
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON O.custid = C.custid
WHERE O.orderid IS NULL;
```

Another is to use the NOT EXISTS predicate, like so:

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE NOT EXISTS(SELECT *
                 FROM Sales.Orders AS O
                 WHERE O.custid = C.custid);
```

Curiously, it appears that currently the optimizer doesn't have logic to detect the outer join solution as actually applying an anti semi join, but it does with the NOT EXISTS solution. This can be seen in the execution plans for these queries shown in Figure 3-19.

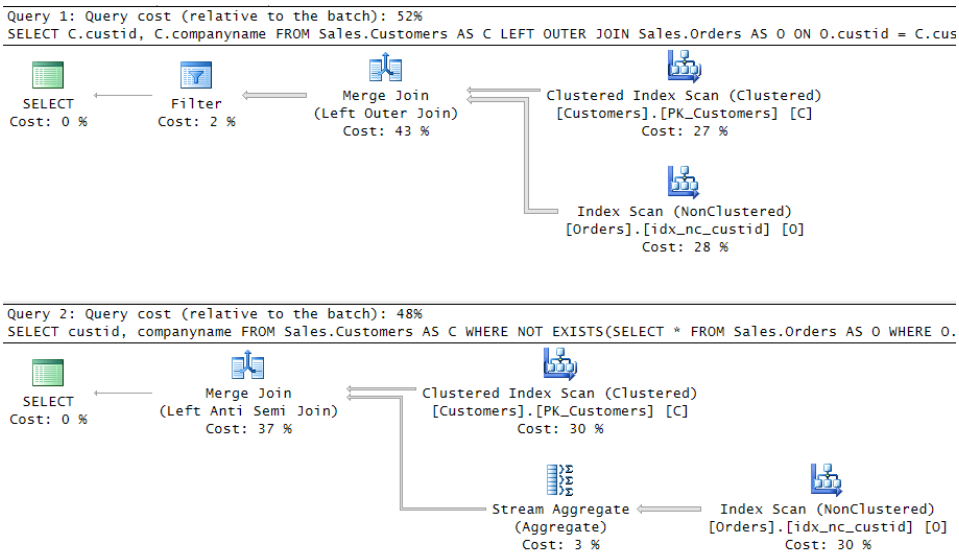

FIGURE 3-19 Plans for anti semi join solutions.

Of course, there are a number of additional ways to implement anti semi joins. The ones I showed are two classic ones.

Join algorithms

Join algorithms are the physical join strategies SQL Server uses to process joins. SQL Server supports three join algorithms: nested loops, merge, and hash. Nested loops is the oldest of the three algorithms and the fallback when the others can't be used. For example, cross join can be processed only with nested loops. Merge and hash require the join to be an equi join or, more precisely, to have at least one join predicate that is based on an equality operator.

In the graphical query plan, the join operator has a distinct icon for each algorithm. In addition, the Physical Operation property indicates the join algorithm (Nested Loops, Hash Match, or Merge Join), and the Logical Operation property indicates the logical join type (Inner Join, Left Outer Join, Left Anti Semi Join, and so on). Also, the physical algorithm name appears right below the operator, and the logical join type appears below the algorithm name in parentheses.

The following sections describe the individual join algorithms, the circumstances in which they tend to be efficient, and indexing guidelines to support them.

Nested loops

The nested loops algorithm is pretty straightforward. It executes the outer (top) input only once and, using a loop, it executes the inner (bottom) input for each row from the outer input to identify matches.

Nested loops tends to perform well when the outer input is small and the inner input has an index with the key list based on the join column plus any additional filtered columns, if relevant. Whether it's critical for the index to be a covering one depends on how many matches you get in total. If the total number of matches is small, a few lookups are not too expensive and therefore the index doesn't have to be a covering one. If the number of matches is large, the lookups become expensive, and then it becomes more critical to avoid them by making the index a covering one.

As an example, consider the following query:

```
USE PerformanceV3;

SELECT C.custid, C.custname, O.orderid, O.empid, O.shipperid, O.orderdate
FROM dbo.Customers AS C
  INNER JOIN dbo.Orders AS O
    ON O.custid = C.custid
WHERE C.custname LIKE 'Cust_1000%'
  AND O.orderdate >= '20140101'
  AND O.orderdate < '20140401';
```

Currently, the tables don't have good indexing to support an efficient nested loops join. Because the smaller side (Customers, in our case) is usually used as the outer input and is executed only once, it's not that critical to support it with a good index. In the worst case, the small table gets fully scanned. The far more critical index here is on the bigger side (Orders, in our case). Nevertheless, if you want to avoid a full scan of the Customers table, you can prepare an index on the filtered column *custname* as the key and include the *custid* column for coverage. As for the ideal index on the Orders

table, think of the work involved in a single iteration of the loop for some customer *X*. It's like submitting the following query:

```
SELECT orderid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE custid = X
  AND orderdate >= '20140101'
  AND orderdate < '20140401';
```

Because you have one equality predicate and one range predicate, it's quite straightforward to come up with an ideal index here. You define the index with the key list based on *custid* and *orderdate*, and you include the remaining columns (*orderid*, *empid*, and *shipperid*) for coverage. Remember the discussions about multiple range predicates in Chapter 2? When you have both an equality predicate and a range predicate, you need to define the index key list with the equality column first. This way, qualifying rows will appear in a consecutive range in the index leaf. That's not going to be the case if you define the key list with the range column first.

Run the following code to create optimal indexing for our query:

```
CREATE INDEX idx_nc_cn_i_cid ON dbo.Customers(custname) INCLUDE(custid);

CREATE INDEX idx_nc_cid_od_i_oid_eid_sid
  ON dbo.Orders(custid, orderdate) INCLUDE(orderid, empid, shipperid);
```

Now run the query and examine the query plan shown in Figure 3-20.

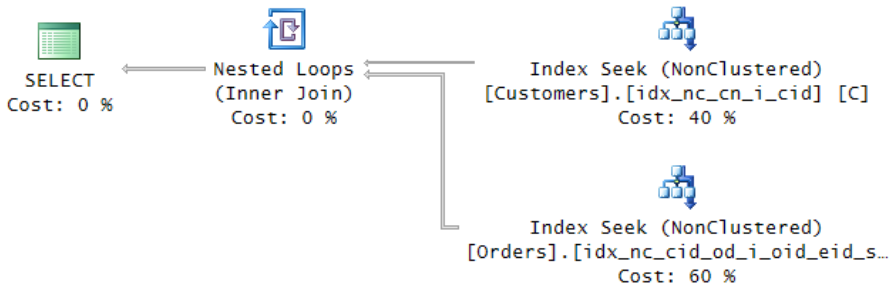

FIGURE 3-20 Plan with a nested loops algorithm.

As you can see, the optimizer chose to use the nested loops algorithm with the smaller table, Customers, as the outer input and the bigger table, Orders, as the inner input. The plan performs a seek and a range scan in the covering index on Customers to retrieve qualifying customers. (There are 11 in our case.) Then, using a loop, the plan performs for each customer a seek and a range scan in the covering index on Orders to retrieve matching orders for the current customer. There are 30 matching orders in total.

Next, I'll discuss the effects that different changes to the query will have on its optimization.

As mentioned, if you have a small number of matches in total, it's not that critical for the index on the inner table to be a covering one. You can test such a case by adding *O.filler* to the query's SELECT