Distributed Hinting

It is not easy to optimise a distributed query, but the most important target, usually, is to minimise the traffic between the databases involved. There aren't many steps you can take to achieve this target, but there is a strategy that tends to help and a hint that is particularly relevant.

Jonathan Lewis, Freelance Consultant, JL Computer Consultancy

Basic Problem

The biggest problem with distributed queries is that the optimizer doesn't seem to allow for the fact that they are distributed, and once it has acquired whatever statistics it can for all the relevant objects it behaves as if all the objects were in the local database. This statement isn't quite true – there are some indications in trace files from 10*g* onwards that the optimizer allows some cost for remote execution of distributed queries, and I've seen occasional clues that it has considered executing distributed queries remotely; but the cost adjustments were very small, and the optimizer seemed to ignore, or fail to use, them.

Let's consider an example: I have a query that joins ***dist_home*** and ***dist_away***, I collect a few rows from dist_home and for each row in dist_home I find some rows in dist_away. I'm likely to see one of two execution plans: a nested loop join with an indexed access from dist_home to dist_away or a hash join with dist_home as the build table and dist_away as the probe table. Here's an example query and the two plans (nested loop first) when dist_away is a remote table.

There's not a lot of difference between these two plans at first sight, but the cost of the nested loop is significantly higher than the cost of the hash join – so

```
select
   dh.small_vc,
   da.large_vc
from
   dist_home                  dh,
   dist_away@orcl@loopback    da
where
   dh.small_vc like '1%'
and       da.id = dh.id;


-------------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Inst   |IN-OUT|
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |   202 | 25250 |   223   (1)|        |      |
|   1 |  NESTED LOOPS      |           |   202 | 25250 |   223   (1)|        |      |
|*  2 |   TABLE ACCESS FULL| DIST_HOME |   202 |  2020 |    19   (0)|        |      |
|   3 |   REMOTE           | DIST_AWAY |     1 |   115 |     1   (0)| ORCL@~ | R->S |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("DH"."SMALL_VC" LIKE '1%')

Remote SQL Information (identified by operation id):
----------------------------------------------------
   3 - SELECT /*+ USE_NL ("DA") */ "ID","LARGE_VC" FROM "DIST_AWAY" "DA" WHERE "ID"=:1
       (accessing 'ORCL@LOOPBACK' )


-------------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Inst   |IN-OUT|
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |   202 | 25250 |    32   (0)|        |      |
|*  1 |  HASH JOIN         |           |   202 | 25250 |    32   (0)|        |      |
|*  2 |   TABLE ACCESS FULL| DIST_HOME |   202 |  2020 |    19   (0)|        |      |
|   3 |   REMOTE           | DIST_AWAY |  2000 |   224K|    13   (0)| ORCL@~ | R->S |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("DA"."ID"="DH"."ID")
   2 - filter("DH"."SMALL_VC" LIKE '1%')

Remote SQL Information (identified by operation id):
----------------------------------------------------
   3 - SELECT /*+ NO_SWAP_JOIN_INPUTS ("DA") USE_HASH ("DA") */ "ID","LARGE_VC" FROM
       "DIST_AWAY" "DA" (accessing 'ORCL@LOOPBACK' )
```

FIGURE 1

```
-------------------------------------------------------------------------------
| Id | Operation              | Name      | Rows | Bytes | Cost (%CPU)| Inst  |IN-OUT|
-------------------------------------------------------------------------------
|  0 | SELECT STATEMENT REMOTE|           |  202 | 44844 |    32   (0)|       |      |
|* 1 |  HASH JOIN             |           |  202 | 44844 |    32   (0)|       |      |
|  2 |   REMOTE               | DIST_HOME |  202 |  3434 |    13   (0)|     ! | R->S |
|  3 |   TABLE ACCESS FULL    | DIST_AWAY | 2000 |   400K|    19   (0)| ORCL  |      |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("A1"."ID"="A2"."ID")

Remote SQL Information (identified by operation id):
----------------------------------------------------
   2 - SELECT "ID","SMALL_VC" FROM "DIST_HOME" "A2" WHERE "SMALL_VC" LIKE '1%'
       (accessing '!' )
```

FIGURE 2

that's the default plan taken in this case. Looking at all the details of operation 3 (REMOTE), though, we can see that really we have to choose the lesser of two evils. (Note that we weren't given much space in the "Instance" column for the database link name for the remote database, and the "IN-OUT" column shows the data flow to be "Remote to Serial").

When we use the nested loop join we have to operate the remote SQL an estimated 200 times – and looking at the Remote SQL Information we see that we're selecting a couple of columns from the remote table for a given ID. That should be a high precision access, so we're looking at a large number of small messages travelling back and fore across the database link.

When we use the hash join we're only going to operate the remote SQL once – but looking at the content of the remote SQL we see that we're selecting all the columns we need from EVERY row in the table. In this case we're going to pull a large volume of (mostly redundant) information across the network (Tip: you can adjust the SQL*Net parameter SDU_SIZE and the O/S network buffer size to make this transfer as efficient as possible).

This is often the problem with distributed queries: which option has the smaller impact on your network, a large number of small round trips or a small number of large round-trips. A choice between latency and throughput.

But there is a third way – if we push the selected dist_home data to the remote site it will be a bulk transfer of a fairly small amount of data; then we can do the join remotely and push the (relatively) small result set back. In this way we have kept both the volume of traffic and the number of messages to a minimum. Choosing which site actually runs the query can make a big difference to performance – and that's what the ***driving_site()*** hint gives us. The hint should reference a table at the site where you want the query to operate; In this case I would add /*+ driving_site(da) */ to my query, which would change the hash join plan into the following (see Figure 2):

Note how the first line of the plan now says "select statement ***REMOTE***". This is the execution plan as seen from the perspective of the remote database. This is why the reference to the DIST_HOME table (operation 2) has become REMOTE; the odd "Instance" identifier of "!" is the name given to the local database when viewed from the perspective of the remote, and notice how the IN-OUT columns now says that it's the DIST_HOME table that is transferred "Remote to Serial".

If we check the numbers in this plan we can see, first of all, that the cost hasn't changed from that of the original hash join (which helps to explain why the optimizer hasn't chosen to make this switch between local and remote). But according to the Bytes column we're going to pull 3,434 bytes to the remote database, do the join, then send 44,844 bytes back – compared to pulling 224KB from the remote database when we executed from the local database. That looks like a potential benefit to me (albeit a small one with this little example.)

Join Order

Unfortunately it's not just the choice of ***where*** to execute the query that matters, and it's possible to run a query from the "right" database but still cause too much network traffic by accessing the tables in the wrong order. Here's a sample query with execution plan (see Figure 3):

We're joining remote tables ***sales*** and ***products*** with local table ***sites***; that being the case we might consider using replication technology to replicate the sites tables (which sounds as if it shouldn't be subject to much change) to the remote database so that we can do a three-table ***remote*** (as opposed to ***distributed***) join. But we're stuck with what we've got at present and what we've got is a plan where Oracle gets

```
select
        sale_date,product, site, qty, profit
from
        sales@&m_target         sal,
        sites                   sit,
        products@&m_target      prd
where
        sit.id = sal.site
and     prd.id = sal.product
and     prd.promoted > date'2014-06-17'
;

-------------------------------------------------------------------------------
| Id  | Operation           | Name     | Rows  | Bytes | Cost (%CPU) | Inst   |IN-OUT|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |          |  3627 |   162K| 10030   (1) |        |      |
|   1 |  NESTED LOOPS       |          |  3627 |   162K| 10030   (1) |        |      |
|   2 |   NESTED LOOPS      |          | 10000 |   322K|    11  (28) |        |      |
|   3 |    REMOTE           | SALES    | 10000 |   263K|     9  (12) | ORCL@~ | R->S |
|*  4 |    INDEX UNIQUE SCAN| SI_PK    |     1 |     6 |     0   (0) |        |      |
|   5 |   REMOTE            | PRODUCTS |     1 |    13 |     1   (0) | ORCL@~ | R->S |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("SIT"."ID"="SAL"."SITE")

Remote SQL Information (identified by operation id):
----------------------------------------------------
   3 - SELECT "SALE_DATE","SITE","PRODUCT","QTY","PROFIT" FROM "SALES" "SAL"
       (accessing'ORCL@LOOPBACK' )

   5 - SELECT "ID","PROMOTED" FROM "PRODUCTS" "PRD" WHERE
       "PROMOTED">TO_DATE(' 2013-06-17 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
       AND "ID"=:1 (accessing 'ORCL@LOOPBACK' )
```

FIGURE 3

every row from the sales table (in array fetches) and checks each row against the sites index then, for each result row individually, accesses the products table. As always there's the question of balancing the number of roundtrips and the volume of data to find the best strategy – in this case I'll suggest that I want to operate locally, get a remote join to take place between products and sales (since this eliminates a lot of data as early as possible), pull the result back in a bulk transfer to the local database then join to sites. In this case all I have to do is put in the hint /*+ leading (prd sal sit) */ (with an optional and currently redundant) driving_site(sit) to get the following plan (see Figure 4).

```
-------------------------------------------------------------------------------------
| Id  | Operation          | Name  | Rows  | Bytes | Cost (%CPU) | Inst   |IN-OUT|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |       |  3627 |   162K|    27   (8) |        |      |
|   1 |  NESTED LOOPS      |       |  3627 |   162K|    27   (8) |        |      |
|   2 |   REMOTE           |       |    45 |   585 |    17   (0) | ORCL@~ | R->S |
|*  3 |   INDEX UNIQUE SCAN| SI_PK |     1 |     6 |     0   (0) |        |      |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("SIT"."ID"="SAL"."SITE")

Remote SQL Information (identified by operation id):
----------------------------------------------------
   2 - SELECT /*+ LEADING ("PRD" "SQL" "SIT") */ "A1"."ID",
       "A1"."PROMOTED","A2"."SALE_DATE","A2"."SITE","A2"."PRODUCT","A2"."QTY",
       "A2"."PROFIT" FROM "PRODUCTS" "A1","SALES" "A2" WHERE "A1"."ID"="A2"."PRODUCT"
       AND "A1"."PROMOTED">TO_DATE(' 2013-06-17 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
       (accessing 'ORCL@LOOPBACK' )
```

FIGURE 4

The cardinality estimate at operation 2 (REMOTE) is clearly wrong, but the optimizer still manages to get a better estimate at the subsequent nested loop operation; and we can see from the Remote SQL Information that the two-table join we wanted to take place at the remote site has indeed occurred as expected.

Strangely, when I had hinted the order as **/*+ leading(sal prd sit) */** – just swapping the order of sal and prd – Oracle used two remote operations to fetch the data from sales and products separately, then joined them locally with a hash join. It doesn't seem reasonable that the optimizer should arrive at this plan, but that's the sort of surprise you can get with distributed queries – even in 12.1.0.2 which is the version I've been using throughout this article.

Distributed DML

When we move from "select" to "create as select" we make a horrid discovery: for no obvious reason the **driving_site()** hint is not valid, so we need to find a different way of dealing with the problem of controlling the query. (Note: this behaviour is not a bug it's deliberate; MoS Bug note 5517609 states: *"This is not a bug. A distributed DML statement must execute on the database where the DML target resides. The DRIVING_SITE hint cannot override this."*)

There are two well-known ways of working around this problem – sometimes it is possible to make the query "efficient enough" by creating suitable join views at the remote site and then querying the view; the alternative, which I will demonstrate here, is to create a pipelined function to hide the select statement. Unfortunately this mechanism can't work with CTAS, or with "insert /*+ append */", so you still suffer a penalty for using distributed queries.

I'm going to use the first query I discussed in the article (joining **dist_home** and **dist_away**) to populate a table in the local database. First I create a table, a scalar type and an array type, then I create a pipelined function that "pipes" rows of the scalar type, then I can write a select statement to insert from the pipelined function into the table (see Figure 5):

You'll notice I've added the "hint" "FIND ME" to the embedded SQL- this was to allow me to search the library cache (***v$sql***) for the statement so that I could find its ***sql_id*** and ***child_number*** and check its execution plan and some of the execution statistics.

Unfortunately the execution plan seemed to disappear the moment the insert completed, so I had to fall back on the extended SQL trace (event 10046 at level 8) to see where each part of the code ran. This allowed me to see that the

```
create table dist_join (small_vc varchar2(10), large_vc varchar2(200));

create or replace type myScalarType as object (
   small_vc          varchar2(10),
   large_vc          varchar2(200)
)
/

create or replace type myArrayType as table of myScalarType
/

create or replace function pipe_fun
return myArrayType pipelined
as
begin
   for r1 in (
           select              /*+
                                       driving_site (da) FIND ME
                               */
                       dh.small_vc,
                       da.large_vc
           from
                       dist_home           dh,
                       dist_away@&m_target da
           where
                       dh.small_vc like '1%'
           and         da.id = dh.id
   ) loop
           pipe row (myScalarType(r1.small_vc, r1.large_vc));
   end loop;
   return;
end;
/

insert into dist_join
select     *
from       table(pipe_fun)
;

commit;
```

FIGURE 5

```
Rows (1st) Rows (avg) Rows (max) Row Source Operation
---------- ---------- ---------- ---------------------------------------------------
      1111       1111       1111 HASH JOIN  (cr=65 pr=0 time=8342 us)
      1111       1111       1111  REMOTE  DIST_HOME (cr=0 pr=0 time=6285 us)
      2000       2000       2000  TABLE ACCESS FULL DIST_AWAY (cr=65 pr=0 time=888 us)

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       15        0.00          0.00
  SQL*Net message from client                     15       68.18         68.19
```

FIGURE 6

embedded query was executed by the remote database, which sent a request for rows from the ***dist_home*** table to the local database and pulled them to the remote database in just two fetch calls. The trace files also showed that the embedded select statement was fetching 100 rows at a time thanks to the standard PL/SQL optimisation of "cursor for loops" – so the execution time was suitably close to standard array processing time and not the row-by-row processing that the PL/SQL appears to be.

Running ***tkprof*** against the trace file from the remote session this (with a few cosmetic cuts) is the information I got for the critical query (see Figure 6).

As you can see, the plan shows that the remote database is, indeed, the driving site for this query. The 15 SQL*Net round-trips also give you some idea of the array-processing efficiency, although you really need to see the details in the trace file to understand exactly where they come from (and why one of them – the last one in the trace file - is 68 seconds).

Summary

It's possible that the optimizer has some code that is supposed to allow distributed queries to execute at a remote site and some code that should recognise that remote access is more expensive than local access, but at present that code doesn't seem to be functioning properly. Because of this we have to tell Oracle when it would be appropriate to execute a query at a remote site.

We may also need to force Oracle into a particular join order to ensure that when two tables are located at the same database the join between them takes place at that database. The ***driving_site()*** hint dictates where the work is done, and the ***leading()*** hint (possibly with the help of a ***no_merge()*** hint) can dictate locality of joins. Unfortunately the ***driving_site()*** hint is not valid as part of either "create as select (CTAS)" or "insert as select", so we have to find an alternative mechanism that allows us to control the distributed query. ■

Footnote

There are other limitations and problems with distributed joins and I've written several articles about the topic on my blog: http://jonathanlewis.wordpress.com/category/oracle/distributed/ For example, one of the common strategies to make remotely joinable tables join remotely is to create a remote view joining then and then query the view; but when you query a remote view, the local optimizer isn't able to move inside the view to discover the statistics of the underlying tables – so joins involving remote views can produce very bad execution plans.

That's not the only feature that results in the optimizer losing information and producing bad execution plans, another is that it does not collect histogram information from remote objects, it only collects the simple column-level statistics. I've even got an example where a four table join does a remote join of the first three tables if the cardinality for the first table is less than 1,000 and switches to three separate remote operations – doing effectively the same sequence of nested loop joins – when the estimated cardinality hits 1,000.

All in all, distributed queries lead to lots of traps and, going back to my opening comment on the difficulties; perhaps there are some bugs in the optimizer that have spent the last few years hiding away waiting for someone to raise the SR that will introduce the fixes that will change everything.

ABOUT THE AUTHOR

Jonathan Lewis
Freelance Consultant, JL Computer Consultancy

Jonathan's experience with Oracle goes back more than 25 years. He specialises in physical database design, the strategic use of the Oracle database engine and solving performance issues. Jonathan is the author of 'Oracle Core' and 'Cost Based Oracle – Fundamentals' both published by Apress and 'Practical Oracle 8i – Designing Efficient Databases' published by Addison-Wesley, and has contributed to three other books about Oracle. He is one of the best-known speakers on the UK Oracle circuit, as well as being very popular on the international scene – having worked or lectured in 50 different countries. Further details of his published papers, presentations and tutorials can be found through his blog: jonathanlewis.wordpress.com.