The other day I gave a presentation on ‘Efficient SQL’. It was the first of a number of presentations, so I started with explaining some basic concepts.

Maybe it will be interesting for other people too. So here is the summary of it.

MAIN RULES OF EFFICIENT SQL

The main rules of efficient SQL are:

- Less is better
- More is better

That seems easy, since everything is better. But let me explain.

Less is better: The less I/O generated the better your statement will perform. Even though somebody might be able to think of some exceptions. It is safe to keep this as a rule of thumb.

Avoid unnecessary work for your query. You can do that by selecting only the rows that you need. (That sounds obvious, but I’ll give an example soon). In complex queries, make sure that you select the smallest possible set in every part of your query. Rather than collecting a huge amount of data and then selecting what you need, select the smallest set possible before you join.

Also select only the columns that you need. One argument is that it might give Oracle a chance to skip the table access, and use an index-only access.

Think about the use of ‘select *’. Most often this is a complete waste of resources. In packaged software, it can easily lead to bugs when the table definition changes. In all situations, it will cost extra resources to collect the data, send them to the client and then filter out the data that is not needed.

More is better: This is of course not about I/O. It is about the information that you give Oracle about your data and your query. Add as many predicates as possible, since it can help the optimizer do a better job. When 2 tables are joined on an ID-column. But you know, from your knowledge of the data, that another column can also be used as a join-condition then use both join conditions.

It will help the optimizer work out the relationship between the tables, and select the optimal plan. If 2 columns have related data, and the predicate on one column means that the other column is restricted too, put a predicate on both columns.

Consider the following:

```
SQL> create table Xxx_inner
```

```
  2  as select * from dba_objects
  3  where object_type='TABLE';

Table created.

SQL> create index xxx_inner_n1 on xxx_inner(object_id,object_type);

Index created.

SQL> create index xxx_inner_n2 on xxx_inner(Object_type);

Index created.

SQL> create table xxx_outer
  2  as select * from dba_objects;

Table created.

SQL> insert into  xxx_outer
  2  select * from dba_objects;

71136 rows created.

SQL> create index xxx_outer_n1 on xxx_outer(Object_id,object_type);

Index created.

SQL> create index xxx_outer_n2 on xxx_outer(object_type);

Index created.

SQL> analyze table xxx_inner compute statistics;

Table analyzed.

SQL> analyze table xxx_outer compute statistics;

Table analyzed.

SQL> set autotrace traceonly;
SQL> select o.*
  2  from xxx_outer o
  3  where o.object_id in (select object_id from xxx_inner i)
  4  and o.object_type='PROCEDURE';

no rows selected

Statistics
----------------------------------------------------------
        113  consistent gets
       1064  bytes sent via SQL*Net to client
```

Consistent gets is the number of times data was read from the buffer cache into our session memory. It is therefore a good measure of the amount of work a session needed to do while executing a query.

```
SQL> select o.*
  2  from xxx_outer o
  3  where o.object_id in (select object_id
                          from xxx_inner i
                          where i.object_type=o.object_type)
  4  and o.object_type='PROCEDURE';
```

```
no rows selected

Statistics
----------------------------------------------------------
         15  consistent gets
       1064  bytes sent via SQL*Net to client

SQL> select o.object_id
  2  from xxx_outer o
  3  where o.object_id in (select object_id
                           from xxx_inner i
                           where i.object_type=o.object_type)
  4  and o.object_type='PROCEDURE';

no rows selected

Statistics
----------------------------------------------------------
         15  consistent gets
        254  bytes sent via SQL*Net to client
```

Here we see two tables created from ‘DBA_OBJECTS’. We join these tables on ‘OBJECT_ID’. In this case, we have some information that the Oracle Optimizer does not have. The object_id,object_type combination is the same in both tables. Unaware of this fact, Oracle has to search all object_id’s for ‘XXX_INNER’. When we tell Oracle that the object_type is the same, it can skip most of the records resulting in less I/O.

When we finally tell Oracle that we’re only interested in the object_id, we also reduce the network traffic by 75%.

The concepts

For every query, Oracle will have to collect some data. And in most cases, it will have to join one or more data-sets. Let’s see what options Oracle has for collecting data and joining the data together. In this presentation we only look at the basic options, not the more sophisticated features. So don’t expect this list to be complete. We will see how data can be retrieved from the database, and how tables / data-sets can be joined together. We will not yet go into the most efficient way to do it, because the most efficient way to retrieve data depends on many factors. Only when you have a basic understanding of the concepts, we can start thinking about the most efficient way to do things.

Collecting data

To gather data from a table, Oracle can use 3 different options, called ‘Access Paths’:

- Full Table Scan
- Index Scan / Table Access by Rowid
- Index Scan

The Full Table Scan (FTS) is exactly as the name implies, a full scan of the table. All the records of the table are read into memory, and checked against the predicates in the query (where clause or Join condition). The blocks of the table don't have to be read in any particular order. Oracle will just try to get the blocks of the table as quickly as possible.

When a sizable part of the table needs to be selected, this will be the most efficient access path. It will be the only one available, if there is no predicate available that matches (part of) an index.

The 'Index Scan / Table Access by Rowid', will use an index to decide which rows need to be read into memory. Then based on the rowid in the index, the correct row will be retrieved. The rowid refers directly to the position on disk where the row is located.

And the last option is the 'Index Scan'. The difference with the previous option is that Oracle does not need to get the table data anymore. The data in the index is sufficient to answer the query.

So which one of these is the most efficient?

As usual, it depends. Many people think a FTS is less efficient than an Index Scan. But consider this:

```
SQL> create table xxx_access as select * from dba_objects;

Table created.

SQL> create index xxx_access_n1 on xxx_access (object_id);

Index created.

SQL> select object_id,object_name from xxx_access where object_id>0;

71139 rows selected.

Elapsed: 00:00:08.02

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=287 Card=78799 Bytes=6225121)
   1    0   TABLE ACCESS (FULL) OF 'XXX_ACCESS' (TABLE) (Cost=287 Card=78799 Bytes=6225121)

Statistics
----------------------------------------------------------
       5803  consistent gets
    2727442  bytes sent via SQL*Net to client
      71139  rows processed

SQL> select /*+ INDEX (xxx_access) */ object_id, object_name from xxx_access where object_id>0;

71139 rows selected.
```

```
Elapsed: 00:00:10.05

Execution Plan
----------------------------------------------------------
0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1290 Card=78799 Bytes=6225121)
1   0   TABLE ACCESS (BY INDEX ROWID) OF 'XXX_ACCESS' (TABLE) (Cost=1290 Card=78799 By=6225121)
2   1     INDEX (RANGE SCAN) OF 'XXX_ACCESS_N1' (INDEX) (Cost=161 Card=78799)

Statistics
----------------------------------------------------------
      10771  consistent gets
    2727442  bytes sent via SQL*Net to client
      71139  rows processed
```

The FTS needs 5803 I/O operations. While the Index Scan takes almost double at 10771 I/O's and 2 seconds more. (out of 10 sec's!)

Imagine what will happen, when you try this with a multi-million row table.

But for sure an Index Scan will be the most efficient when it can be done? Again, it depends. Consider this:

```
SQL> create table test1 (l number, txt varchar2(500));
Table created.

SQL> create index test1_idx on test1(l,txt);
Index created.

SQL> Begin
  2  For I In (Select  Level L, Rpad('ABC',500,To_char(Level)) Txt
  3            From Dual Connect By Level <= 50000) Loop
  4    Insert Into Test1 Values (I.L,I.Txt);
  5    If Mod(I.L,5)!=0 Then
  6       Delete From Test1 Where L=I.L;
  7    End If;
  8  End Loop;
  9  Commit;
 10  end;
 11  /
PL/SQL procedure successfully completed.

SQL> analyze table test1 compute statistics;
Table analyzed.

SQL> set autotrace traceonly;
SQL> select l from test1 t where l>0;
10000 rows selected.

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=773 Card=10000 Bytes=40000)
   1    0   TABLE ACCESS (FULL) OF 'TEST1' (TABLE) (Cost=773 Card=10000 Bytes=40000)

Statistics
----------------------------------------------------------
       1393  consistent gets
     140549  bytes sent via SQL*Net to client
      10000  rows processed
```

```
SQL> select /*+ INDEX(t,test1_idx) */ 1 from test1 t where 1>0;
10000 rows selected.

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3338 Card=10000 Bytes=40000)
   1    0   INDEX (RANGE SCAN) OF 'TEST1_IDX' (INDEX) (Cost=3338 Card=10000 Bytes=40000)
Statistics
----------------------------------------------------------
       4003  consistent gets
     140549  bytes sent via SQL*Net to client
      10000  rows processed
```

How can the Full Table Scan be more efficient?

Good question. To answer it, we have to look at the amount of work that Oracle has to do to get the data. We start with the FTS:

As mentioned, a FTS needs to scan all the blocks in a table. In my system, the table XXX_ACCESS was created with 1152 blocks. So basically Oracle will read 1152 blocks. However Oracle has optimized this process. One of the most noticeable optimizations is the db_file_multiblock_readcount, which tells Oracle to read multiple blocks in one I/O operation. So instead of doing 1152 reads, Oracle reads 4 (on my system) blocks at a time in 288 reads. (Take a look at the explain plan above again, and notice the cost of the Full Table Scan!).

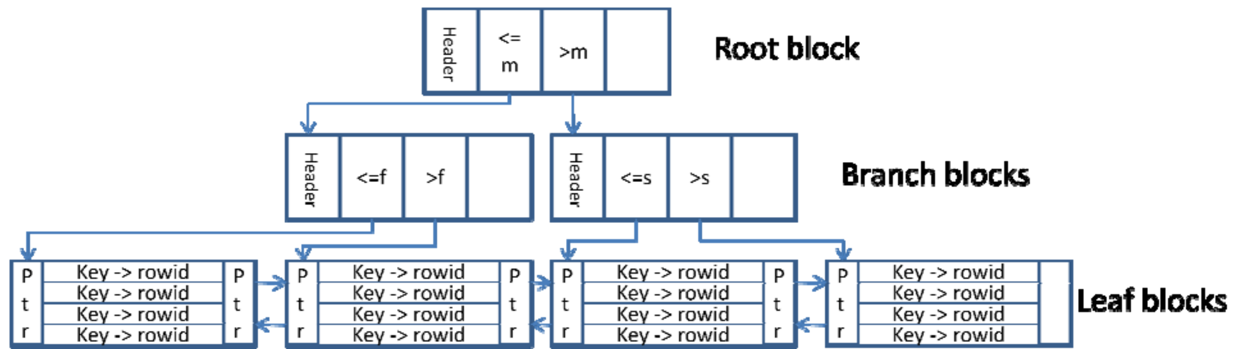
Now how much work does an Index Scan / Table Access rowid need to do?

To answer that question, you need to know the structure of an index. An index in Oracle is a B-Tree structure. It looks like an inverted tree, with the top being the 'Root-block'. The lowest level contains the 'Leaf blocks', that hold the index keys and the matching rowid's. The index keys in the leaf blocks are sorted. So the lowest value will be in the utter-left block, the highest in the far right block.

All upper level blocks show the ranges that the lower level blocks contain.

To find a range of data, Oracle starts at the root block, and follows the pointers to the first leaf block containing an index key within the range. From here Oracle can walk the leaf blocks from left to right (or right to left, if needed).

See the following picture:



That means that Oracle has to walk the 'height' of the index (Index level). Then read a number of leaf blocks. And for every entry in the leaf-block, it needs to retrieve the data from the table.

In the worst case scenario, the data is spread throughout the table. And for every index key, Oracle has to retrieve a different table block. In that case, the amount of work is: (index level – 1) + number of index leaf blocks + (Number of keys * Number of table blocks).

It will be obvious that is a lot more I/O than just reading the table once. The formula is not completely correct, because Oracle does not read the same table block twice, when the next rowid from the index is in the same block.

But the formula should make it clear that with more data being retrieved, the cost of the index access / Table Rowid, is increasing faster than the cost of the Full Table Scan.

What happened to Index Only Access?

The example showed that a Full Table Scan is still more efficient than using only an index. Even though Oracle would only have to walk through the index to retrieve all the data.

This is caused by a feature of the index structure and the way the data was entered into the table. This caused the index to grow bigger than the actual table:

```
SQL> select table_name,num_rows,blocks from dba_Tables where table_name='TEST1';

TABLE_NAME             NUM_ROWS     BLOCKS
-------------------- ---------- ----------
TEST1                     10000        772

SQL> select index_name, distinct_keys, leaf_blocks from dba_indexes where index_name='TEST1_IDX';

INDEX_NAME           DISTINCT_KEYS LEAF_BLOCKS
-------------------- ------------- -----------
TEST1_IDX                    10000        3334
```

This situation can occur when a lot of inserts and deletes are taking place in the same transaction. The space for the deleted table rows can be reused immediately. But the space for the deleted index keys only comes available after the commit. NOTE: The space in the index will be reusable after the commit!

Another situation where this can happen is with an index key based on a sequence. So new data is only inserted at the end of the range. When you delete a lot of values on the lower end of the range (but not all). Richard Foote has some excellent material on his website about Deleted Index Keys:

http://en.wordpress.com/tag/index-delete-operations/

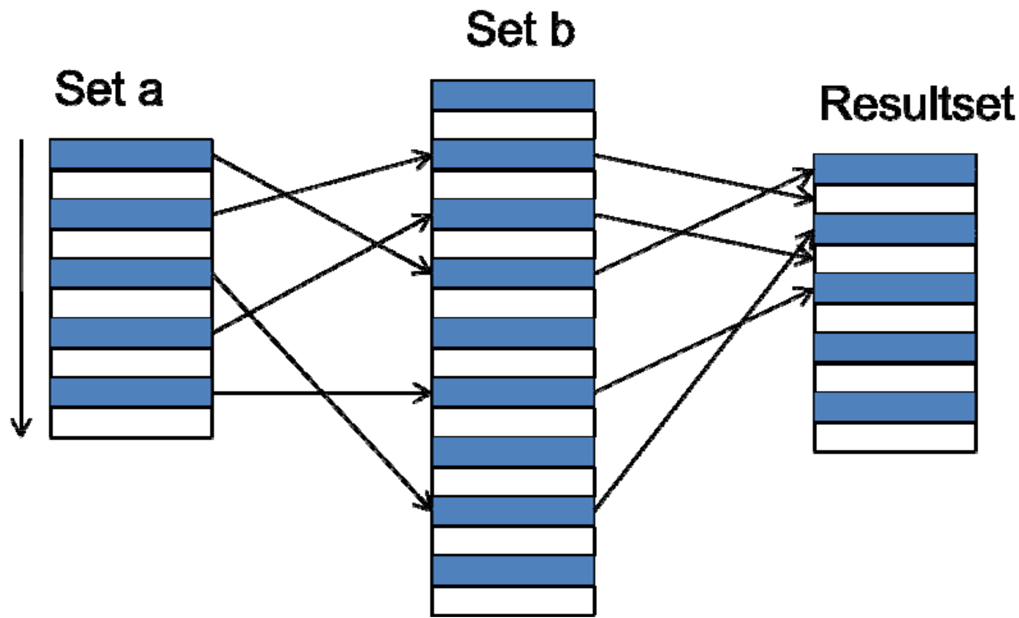
Join Mechanisms

I hope you're not yet in despair. Because so far we have only retrieved data from single tables. In most cases, we need to join tables together to get our data. Now we will look at 3 basic forms of joining data-sets together. Note that it is not necessarily only tables that we join. It can also be a result set from an earlier part of your query. For example we might collect some data from an Index Only Access and then join it to data from a Full Table Scan.

Oracle has 3 mechanisms to perform Joins: Nested Loops, Hash Join, Merge Join. Let's take a look at them.

Nested Loops

The Nested Loop join loops over a smaller data set, and for every record in that set, it searches a matching record in the second data set. Visually, it looks like this:



You can see that the first set is fully scanned. This is the Outer or Driving Set. For every record in this set, we look up a matching record in the second set. Usually this will be done through an index, even though this is not mandatory.

The efficiency of this mechanism depends on the size of Set A, and the number of records we need to retrieve from Set B.

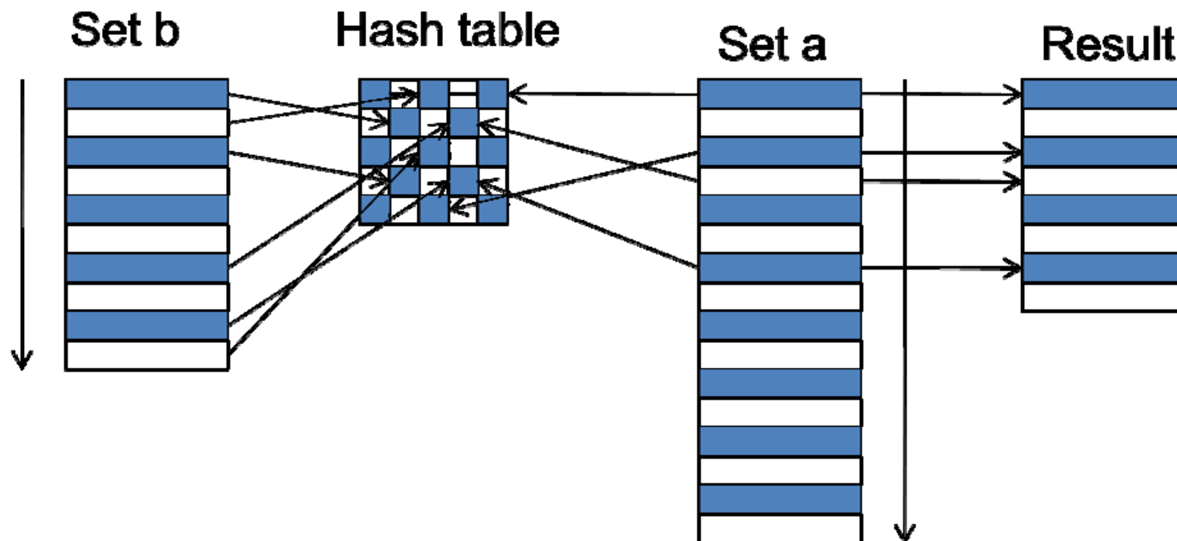
We will see some situations where a Nested Loops join is more or less efficient.

The second join mechanism is the

Hash Join

For a Hash Join, Oracle builds a hash table from a (preferably) smaller data set (Build set), where the key is a hash value derived from the join columns. Then Oracle reads the second data set (Probe set), derives the hash value on the join columns and probes the hash table for a match.

Schematically, it looks like this:



For the Hash Join, it is not relevant if the data is retrieved through an index, or from a Table scan. Both data sets need to be read fully. One of the features of hashing is that collisions may occur. A collision means that 2 different values result in the same hash value. Therefore, the full data set is stored in the hash table. When a match is found on the hash values, Oracle double-checks the actual values of the join columns to see if they match.

There is a major drawback for this mechanism. When the hash table does not fit into the available memory (hash_area_size). Then Oracle will dump parts of the hash table to disk (Temp tablespace). A bitmap of all possible hash values is kept in memory with a bit indicating if a hash-value is used or not.

So while scanning the Probe set, Oracle can check if there is a possible match. If that part of the hash table is not in memory, Oracle will set aside the records from the probe table. When finished with the probe set, the next part of the hash table is loaded into memory, and the records that were set aside are tested again. This is called a 'Multipass Hash Join', instead of the 'Onepass Hash Join' where the entire hash table is held in memory.

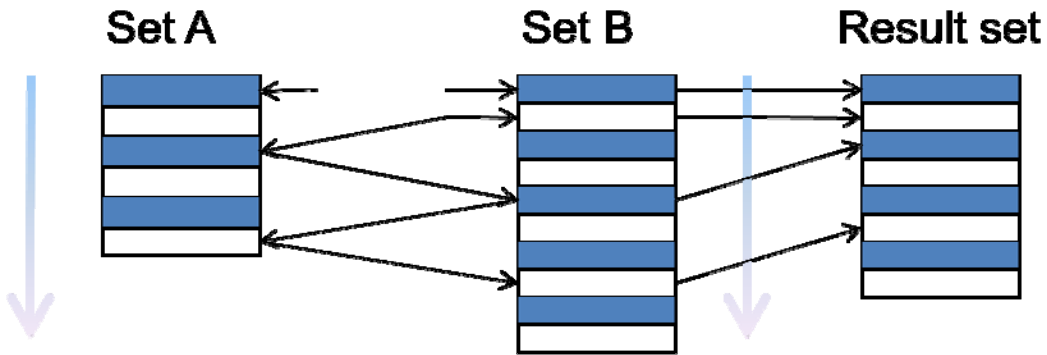
You can imagine that the hash join can be very efficient even without index access. However it can deteriorate quickly when a 'Multipass Hash Join' is needed.

That brings us to the

Merge Join

A Merge Join is possible when 2 datasets are both ordered on their join columns. The datasets can then be read in an alternating way. You start reading the first dataset, then you read the second dataset until you find a matching value, or exceed the value. If it is a matching value, you can start building your result set. If you pass the join value, you continue reading the first dataset again.

Schematically, it looks like this:



The main requirement is of course that both data sets are ordered, before the join takes place. If that is the case, this is probably the most efficient join mechanism. It can handle all kinds of join comparisons, including range comparisons.

Back to efficient SQL

Now it is time to go back to the main focus of this article. How to write efficient SQL.

We have seen how Oracle can access data and join it together. It is the job of the Oracle Optimizer to find out the most efficient way to do that, for a given query. It does this based on statistics on the tables and indexes. It is not in the scope of this article to discuss how to gather statistics. But we will see the use of several of the statistics. These statistics include (but are not limited to) the number of rows in the table, the number of blocks, the average row length, etc. For an index, they include (but are not limited to) the number of leaf blocks, the number of keys per leaf block, the number of datablocks in the table per key, etc.

In an ideal world, we would be able to trust the optimizer to do the right thing all the time.

However, we live in an imperfect world, and the Oracle Optimizer does not always have perfect information about the data or your query. Either the statistics might not be up to date, or they might not include some dependencies within the data. Also your query might not give the optimizer all the information that you have. (see the first example in this article, where we can give the optimizer extra information by adding a predicate).

Consider this query:

```
Select * From
Tiny  T, --7089 rec
Small S   -- 71151 Rec
Where t.Object_id = s.Object_id;
```

Which explain plan is more efficient:

```
0 SELECT STATEMENT
1  HASH JOIN
2   TABLE ACCESS FULL  TINY
3   TABLE ACCESS FULL  SMALL
```

Or:

```
0 SELECT STATEMENT
1  TABLE ACCESS BY INDEX ROWID SMALL
2   NESTED LOOPS
3    TABLE ACCESS FULL          TINY
4    INDEX RANGE SCAN           SMALL_IDX
```

The first explain plan uses a Hash Join, with Tiny as the Build Table. The second uses a Nested Loops Join, with Tiny as the Inner Table.
I hope you have decided that you don't know which one is the most efficient. Because you cannot know based on the information you have. If I selected 2 non-overlapping sets of data, the Nested Loop Join might be more efficient. It would scan Tiny for 7089 records, then do 7089 index lookups on Small (without result), so it would need to read approximately 7089+7089=14178 blocks.

If the 2 sets would have a 1-n relationship, and every occurrence of object_id in Small is also in Tiny, both tables would need to be read fully and the Hash Join would be more efficient.

That means that to write efficient SQL, you'll need to have an understanding of the data. And you must have considered the optimal execution plan for your query.

When you write your query, keep in mind the mantra in first part of this article. Less is Better, More is Better. You want to give Oracle as much information as possible, and you want to get as small as possible result sets.

The last part that we will do in this part, is to look at the

Explain plan

To see what Oracle will do when executing a query, you can make an explain plan. Many tools have built-in options to show explain plans on queries. Alternatively, you can use the Oracle commands:

```
SQL> Explain plan for
  2  Select * From
  3  Tiny  T, --7089 rec
  4  Small S   -- 71151 Rec
  5  Where t.Object_id = s.Object_id;

Explained.

SQL> select * from table( dbms_xplan.display() );
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 803478362

----------------------------------------------------------------------------
| Id  | Operation          | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |       |  7089 |  1287K|  1264   (1)| 00:00:16 |
|*  1 |  HASH JOIN         |       |  7089 |  1287K|  1264   (1)| 00:00:16 |
|   2 |   TABLE ACCESS FULL| TINY  |  7089 |   650K|   213   (0)| 00:00:03 |
|   3 |   TABLE ACCESS FULL| SMALL | 71151 |  6392K|  1049   (1)| 00:00:13 |
----------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("T"."OBJECT_ID"="S"."OBJECT_ID")

15 rows selected.
```

I took the previous query, and ran an explain plan on it. First I should tell you how I created tiny and small, so you can verify:

```
create table tiny as select * from dba_objects where mod(object_id,10)=0;

create table small as select * from dba_objects;

create index tiny_idx on tiny(object_id);

create index small_idx on small(object_id);

Analyze Table Tiny Compute Statistics;

analyze table small compute statistics;
```

You can see that both tables were created from dba_objects and tiny is a subset of small. Therefore a Hash Join makes most sense.

Let’s take a closer look at the explain plan now. Keep in mind that all the numbers are estimates from the optimizer. They are based on the statistics available, and the runtime numbers might be completely different.
The execution starts at the rows furthest right. And top down. In this case that means that we start with a Full Table Scan of Tiny. Then we do a Full Table Scan of Small, and finally we Hash join the 2 sets together.
The result of the Hash Join is returned, and becomes the result of the query (Select).

Let’s take a closer look at the Full Table Scans. After the name of the table, we see ‘Rows’,’Bytes’,’Cost’ and ‘Time’.
The ‘Time’ is an estimate of the amount of time it will take to complete this step. It is useful in query tuning to see which step will take the most time.

The ‘Bytes’ are an estimate of the amount of data used to complete this step. I find it useful when a hash join is involved. Because when it exceeds my hash_area_size, Oracle will need a ‘Multi-Pass Hash Join’.
The ‘Cost’ is often used for query tuning, and people will try to ‘tune down’ the cost. This makes sense, because the cost is an estimate for the amount of I/O Oracle needs to do for this step. However, there is a logical trap in this.
Oracle has used the cost of different explain plans for the original query to decide the most efficient one. Now when we change the query, the cost can no longer be compared to the original query. After all, it is a different query. It will return a different result. Unless, of course there is some information about the data we are selecting, that we didn’t give the optimizer initially.

To me, it makes more sense to look at the ‘Rows’. This is the number of records Oracle expects from that step in the query. If this is very different from the number you are expecting, something is wrong. Either the statistics are outdated, or the Optimizer is missing some information that you have, or you are not selecting the data you are expecting.

In the query above, we see that Oracle has made the perfect assumptions. Tiny will return 7089 rows, Small will return 71151 rows and there is a 1-1 relationship between them. So the join will return one record for each record in Tiny. No use looking for a more efficient plan here. Unless again, we did not query what we are looking for.