U.P.B. Sci. Bull., Series C, Vol. 68, No. 4, 2006

THE OPTIMIZATION OF DATA ACCESS USING "JOIN" CLAUSE

I. RUSU, C. ARITON, Mihaela CĂPĂTOIU, Vlad GROSU[*]

În acest articol vom analiza cuplajele (join) din clasa 'asocierilor interne', folosite în mod curent de către Oracle 9i, şi le vom urmări comportamentul în diferite situaţii, cu ajutorul cheilor (hints) şi a planurilor explicative (explain plan). În mod normal, comportamentul constă din aceea că optimizorul estimează un cost al fiecărui plan de execuţie conform cu căile de acces existente şi a tipurilor de asocieri disponibile, bazându-se pe statisticile aplicate tabelelor, indecşilor sau altor factori.

Urmând aceste consideraţii teoretice, articolul de faţă prezintă câteva exemple de analiză originale, care conduc către concluzii practice şi demne de urmat.

In this article we will present an analysis of the 'inner type' of joins, used in Oracle 9i, and also of their behavior in different cases, using hints and explain plans. Generally, the behavior consists in that the optimizer estimates a cost for each execution plan according to the available access paths and types of join methods, using the statistics on tables, indexes or other factors.

Following these theoretical facts the present article shows several original analysis examples, leading to practical and 'to be followed' conclusions.

Keywords: SQL, clause, join, hint, project plan, algorithm, efficiency.

Introduction

The relational databases have two types of operators: the *set operators* and the *specific operators*. The operator that generates an extended amount of time for a query is the *join operator* and it belongs to *specific operators'* class.

Suppose θ is an arithmetic operator, *x* an attribute of the relation *A* and *y* an attribute of the relation *B*, where the attributes are defined on compatible domains.

Definition. *θ* is called a join of the relations *A* and *B* by the attributes x, y. It has the form:

A|X|B
x θ y

[*] Prof., Dept. of Electronic Technology and Reliability, Faculty of Electronics, University "Politehnica" of Bucharest, Database Architect, Dept. Head of Billing, Crisoft, Assist., Assist., Dept. of Electronic Technology and Reliability, Faculty of Electronics, University "Politehnica" of Bucharest, ROMANIA

and represents a relation *R* that contains the tuples of the Cartesian product *A*X*B*, with the property that all the values of the attributes *x* and *y* respect the *xθy* relation. The Cartesian product of the relations *A* and *B* represents a relation that contains the tuples resulting from the concatenation of each tuple within relation *A* with each tuple within relation *B*.

Due to the high level of the relations' Cartesian product, most of the time we are wondering:

- which of the used *join* methods offers more performance?,
- which are the algorithms they are build upon,
- how can we choose the optimal algorithm,
- how can we affect the process of choosing a different algorithm than the optimizer has already chosen, by default ?

This article will primarily focus on these issues, which will be clarified in the following.

1. Comparison between different *join* methods in *Oracle 9i* using the *hints* and the *explain plan*

We will make an analysis of the *inner type* of joins, used in *Oracle 9i* and will follow their behavior in different cases using hints and explain plan. Generally, the optimizer estimates a cost for each execution plan according to the available access paths and types of join methods, using the statistics on tables, indexes or other factors. The optimizer compares the costs of the explain plans and chooses the one with the lowest cost. Other important factors in query optimization are *throughput* (CPU, memory, physical blocks reads etc) and the *response time*. To find which of the join methods is faster and more efficient, we use the *hints*.

Provided as example, the following query suggests the relationship between two tables, namely *software_table* having the following structure: *soft_id*, *name_soft*, *version*, *size_soft*, *vendor_id*, and *vendor_table* comprising on *vendor_id*, *name_vendor*, *country_vendor*.

```
select name_soft, version,size_soft, name_vendor
from software_table a, vendor_table b
where a.vendor_id=b.vendor_id;
```

The indexes in the two tables are *soft_id_pk* and *vendor_id_pk*, respectively. The relation between the two tables is achieved through *vendor_id* column.

The study will start with two cases in accordance with the number of records in the tables. In the first case, the *software_table* and *vendor_table* tables will be populated with 100,000 and 10,000 records, respectively, and in the second case they will be populated with 1,000,000 and 100,000 records, respectively.

In order to display the execution plan based on *cost* - referred to as *cost based optimization* (*CBO*), we'll analyze the tables using the command analyze table <*name_table*> compute statistics, set explain to "on" as in *autotrace* on and then run the query above.

Before finding what join the optimizer has chosen by default, let us study the fundamental types of algorithms that perform a join operation. There are three main types of algorithms: *nested-loops* join, *sort merge* join and *hash* join.

1.1. The *Nested-Loops* algorithm used in *join* operation

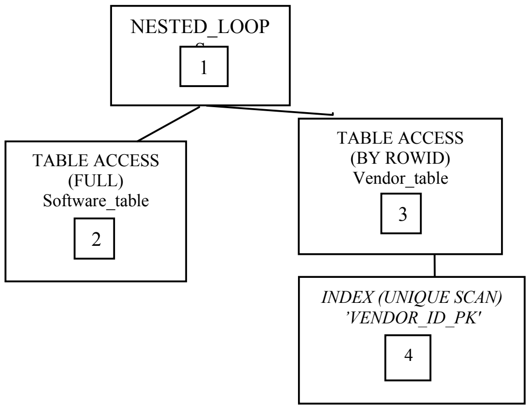
Nested-Loops join is a preferred algorithm for simple queries. The optimizer chooses one table and names it *outer table*, the other one being the *inner table*.

For each tuple in the *outer join* relation, the entire *inner join* relation is scanned and any tuples that match the join condition are added to the result set. *Oracle* combines the data for each tuple set that satisfies the join condition and it will display the resulting tuples.

The pseudo code of this algorithm is:

```
For Each outer_tablerow in outer_table
 For Each key in inner_table.primary_key_index
  If outer_tablerow equals key
      Fetch inner_tablerow from inner_table
      Combine columns from outer_tablerow and inner_tablerow
      Return combined row to client
  Exit For
  End If
End For;
```

In our case, the execution plan using the *nested-loops join* algorithm is depicted in the Fig. 1.1.


NESTED_LOOPS
1
TABLE ACCESS (FULL) Software_table
2
TABLE ACCESS (BY ROWID) Vendor_table
3
INDEX (UNIQUE SCAN) 'VENDOR_ID_PK'
4
Fig.1.1 - Nested-Loops

The optimization module based on cost of the explain plan for nested-loops join is obtained using *USE_NL* hint like below:

```
select /*+USE_NL(a,b)*/
        name_soft, version,size_soft, name_vendor
 from software_table a, vendor_table b
 where a.vendor_id=b.vendor_id;
```

In the first case, we have the following execution plan:

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=100111
Card=100000 Bytes=12100000)
   1    0   NESTED LOOPS (Cost=100111 Card=100000 Bytes=12100000)
   2    1     TABLE ACCESS (FULL) OF 'SOFTWARE_TABLE' (Cost=111
Card=100000 Bytes=6800000)
   3    1     TABLE ACCESS (BY INDEX ROWID) OF 'VENDOR_TABLE' (Cost=1
Card=1 Bytes=53)
   4    3       INDEX (UNIQUE SCAN) OF 'VENDOR_ID_PK' (UNIQUE)
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
     214405  consistent gets
        273  physical reads
          0  redo size
   12690964  bytes sent via SQL*Net to client
      73825  bytes received via SQL*Net from client
       6668  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
   100000  rows processed
Elapsed: 00:09:48
```

In the second case, the execution plan is:

```
Execution Plan
--------------------------------------------------------
0   SELECT STATEMENT Optimizer=CHOOSE (Cost=1001090 Card=1000000
Bytes=125993826)
1   0   NESTED LOOPS (Cost=1001090 Card=1000000 Bytes=125993826)
2    1     TABLE ACCESS (FULL) OF 'SOFTWARE_TABLE' (Cost=1139
Card=1000000 Bytes=70996521)
3    1     TABLE ACCESS (BY INDEX ROWID) OF 'VENDOR_TABLE' (Cost=1
Card=1 Bytes=55)
4    3     INDEX (UNIQUE SCAN) OF 'VENDOR_ID_PK' (UNIQUE)

Statistics
--------------------------------------------------------
          0 recursive calls
          0  db block gets
    2144301  consistent gets
      11828  physical reads
          0  redo size
  130829865  bytes sent via SQL*Net to client
     733792  bytes received via SQL*Net from client
      66665  SQL*Net roundtrips to/from client
          0 sorts (memory)
          0  sorts (disk)
1000000  rows processed
Elapsed: 01:09:52.01
```

Generally, *Nested Loops Join* is used where a small and large row sets are joined but also where two small row sets are joined.

This algorithm is inefficient when the indexes from tables are missing or if the indexed criteria are not very selective.

1.2. The *Hash Join* algorithm used in *join* operations

Hash Join is a faster method than *Nested-Loops Join* suitable for the situations when the indexes are missing or when the criteria are not very selective, being used only in the *equijoin* types. In the hash join, *Oracle* reads all the records from the smaller table, builds a *hash table*, and then reads the larger table to probe which of the records in the hash table are matching, using a key called *hash key*.

The *HASH_AREA_SIZE* configuration parameter determines the size of the table placed into cache. When the available cache space exceeded, hash tables are split apart and temporarily stored in a sorting space. This is detrimental for performance.

The pseudo code of this algorithm is:

```
For Each small_table_row in small_table
        Add small_table_row to hash (keyed on small_table.col)
End For
For Each large_table_row in large_table
        Look up small_table_row from hash using large_table. small_table_row
        as key
        Combine columns from large_table_row and small_table_row
        Return combined row
End For;
```

The execution plan using *hash join* is presented in the Fig. 1.2.

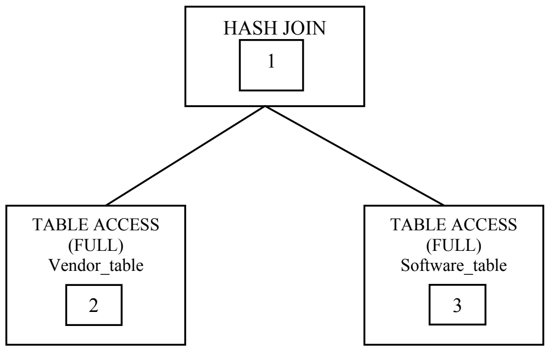

HASH JOIN
1
TABLE ACCESS (FULL) Vendor_table
2
TABLE ACCESS (FULL) Software_table
3

Fig.1.2 - Hash Join

The optimization module based on execution cost for *hash join* is obtained using *USE_HASH* hint in this way:

```
select /*+USE_HASH(a,b)*/
        name_soft, version, size_soft, name_vendor
 from software_table a, vendor_table b
 where a.vendor_id=b.vendor_id;
```

In the first case we have the following execution plan:

```
Execution Plan
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=CHOOSE (Cost=335 Card=100000
Bytes=12100000)
1   0   HASH JOIN (Cost=335 Card=100000 Bytes=12100000)
```

```
2   1   TABLE ACCESS (FULL) OF 'VENDOR_TABLE' (Cost=11 Card=10000
Bytes=530000)
3       1        TABLE ACCESS (FULL) OF 'SOFTWARE_TABLE' (Cost=111
Card=100000 Bytes=6800000)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       7841  consistent gets
        296  physical reads
          0  redo size
   12690964  bytes sent via SQL*Net to client
      73825  bytes received via SQL*Net from client
       6668  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
     100000  rows processed
Elapsed: 00:00:29.08
```

In addition, the optimizer chooses by default this algorithm for our query because the cost and the execution time are the smallest.

When the algorithm is chosen by default it is not necessary to use the hint. The hint is used for displaying the execution plan for others algorithms that are not chosen by default.

In the second case, we have the following execution plan:

```
Execution Plan
----------------------------------------------------------
   0       SELECT STATEMENT Optimizer=CHOOSE (Cost=3436 Card=1000000
Bytes=125993826)
   1    0   HASH JOIN (Cost=3436 Card=1000000 Bytes=125993826)
   2       1          TABLE ACCESS (FULL) OF 'VENDOR_TABLE' (Cost=99
Card=100000 Bytes=5500000)
  3      1        TABLE ACCESS (FULL) OF 'SOFTWARE_TABLE' (Cost=1139
Card=1000000 Bytes=70996521)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      12854  consistent gets
      23963  physical reads
          0  redo size
  130829766  bytes sent via SQL*Net to client
```

```
    733792  bytes received via SQL*Net from client
     66665  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
    1000000 rows processed
Elapsed: 01:07:15.05
```

The *hash join* algorithm is useful either when we have a large table (e.g. over 10,000 records) in relation with a smaller table, or when both tables are very large and without indexes.

1.3. The *Sort Merge Join* used in *join* operations

This algorithm is executed in three steps. The first two steps sort both tables separately and the third step merges them together into a sorted result. Typically, the sort merge join fits the situations when no indexes are used on any table or when the sorting is imposed by using ORDER BY or GROUP BY clauses.

The pseudo code of this algorithm is:

```
function sortMerge(relation left, relation right, attribute a)
    var relation output
    var list left_sorted := sort(left, a)
    var list right_sorted := sort(right, a)
    var left_key
    var right_key
    var set left_subset
    var set right_subset
    advance(left_subset, left_sorted, left_key, a)
    advance(right_subset, right_sorted, right_key, a)
    while not empty(left_sorted) and not empty(right_sorted)
        if left_key = right_key
            add cross product of left_subset and right_subset to output
            advance(left_subset, left_sorted, left_key, a)
            advance(right_subset, right_sorted, right_key, a)
        else if left_key < right_key
            advance(left_subset, left_sorted, lef_key, a)
        else right_key < left_key
            advance(right_subset, right_sorted, right_key, a)
    return output

function advance(key, subset, sorted, a)
    key = sorted[1].a
    subset = emptySet
```

```
while not empty(sorted) and sorted[1].a = key
   insert(subset, sorted[1])
   remove first element from sorted;
```

The execution plan using sort *merge join* is presented in the figure below (see Fig. 1.3).

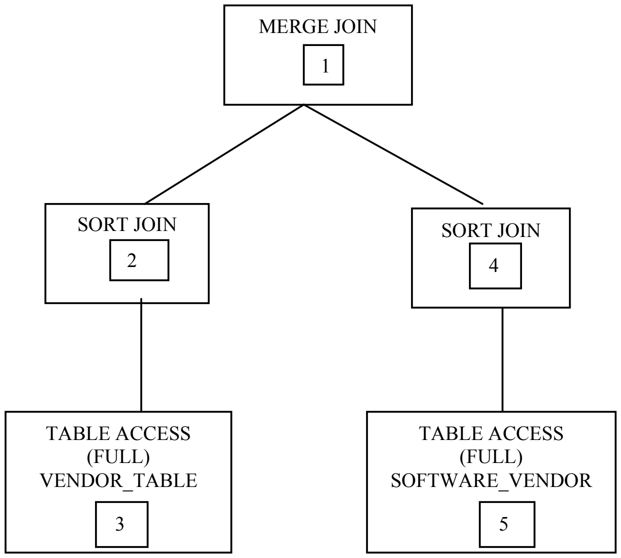

MERGE JOIN
1
SORT JOIN
2
SORT JOIN
4
TABLE ACCESS (FULL) VENDOR_TABLE
3
TABLE ACCESS (FULL) SOFTWARE_VENDOR
5

Fig.1.3 - Sort Merge Join

For the sort merge join, the optimization module based on the execution cost is obtained using *USE_MERGE* hint, like this:

```
select /*+USE_MERGE(a,b)*/
          name_soft, version, size_soft, name_vendor
 from software_table a, vendor_table b
 where a.vendor_id=b.vendor_id;
```

In the first case, we have the following execution plan:

```
Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE (Cost=3092 Card=100000
Bytes=12100000)
   1   0   MERGE JOIN (Cost=3092 Card=100000 Bytes=12100000)
   2   1     SORT (JOIN) (Cost=118 Card=10000 Bytes=530000)
   3   2       TABLE ACCESS (FULL) OF 'VENDOR_TABLE' (Cost=11
Card=10000 Bytes=530000)
   4   1     SORT (JOIN) (Cost=2975 Card=100000 Bytes=6800000)
   5   4       TABLE ACCESS (FULL) OF 'SOFTWARE_TABLE' (Cost=111 Card
=100000 Bytes=6800000)
```

```
Statistics
----------------------------------------------------------
          0  recursive calls
         17  db block gets
       1242  consistent gets
       2616  physical reads
          0  redo size
    8150598  bytes sent via SQL*Net to client
      73825  bytes received via SQL*Net from client
       6668  SQL*Net roundtrips to/from client
          1  sorts (memory)
          1  sorts (disk)
     100000  rows processed
     Elapsed: 00:00:34
```

In the second case, the execution plan is:

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=65008
Card=1000000 Bytes=125993826)
   1    0   MERGE JOIN (Cost=65008 Card=1000000 Bytes=125993826)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'VENDOR_TABLE'
(Cost=1217 Card=100000 Bytes=5500000)
   3    2       INDEX (FULL SCAN) OF ' VENDOR_ID_PK ' (UNIQUE)
(Cost=209 Card=100000)
   4    1     SORT (JOIN) (Cost=63791 Card=1000000 Bytes=70996521)
   5    4       TABLE ACCESS (FULL) OF 'SOFTWARE_TABLE' (Cost=1139
Card=1000000 Bytes=70996521)
```

```
Statistics
----------------------------------------------------------
          0  recursive calls
        123  db block gets
      13842  consistent gets
      46234  physical reads
          0  redo size
   82636602  bytes sent via SQL*Net to client
     733792  bytes received via SQL*Net from client
      66665  SQL*Net roundtrips to/from client
          0  sorts (memory)
          1  sorts (disk)
    1000000  rows processed
Elapsed: 01:08:15.02
```

This algorithm suits for larger data sets already sorted, where the *hash join* and *nested-loops join* algorithms cannot be applied. As we can notice in this study (see Figs. 1.4 and 1.5 below), the number of records has an important role in choosing the algorithm's type. In addition, this method based on execution cost, called *CBO*, depends on many parameters as you can see in the execution plan (db block gets, memory, disk, CPU etc). Consequently, the optimizer makes a compromise between the response time and throughput.

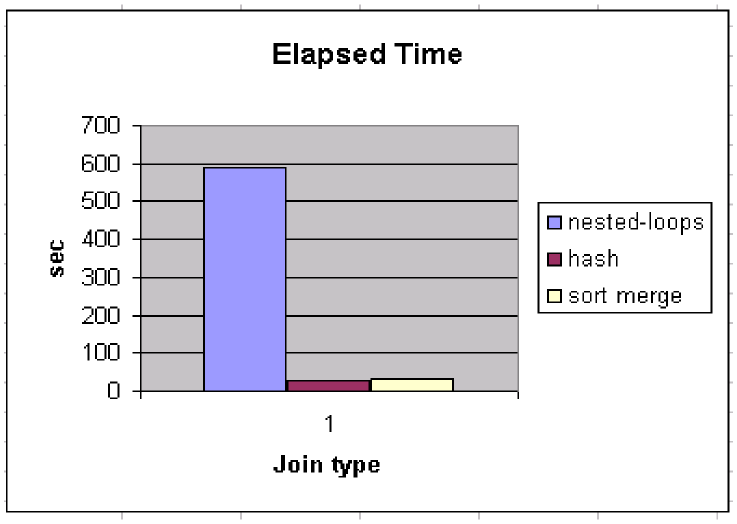


Fig 1.4 - The response time *with fetching data* for all the algorithms, in the first case.

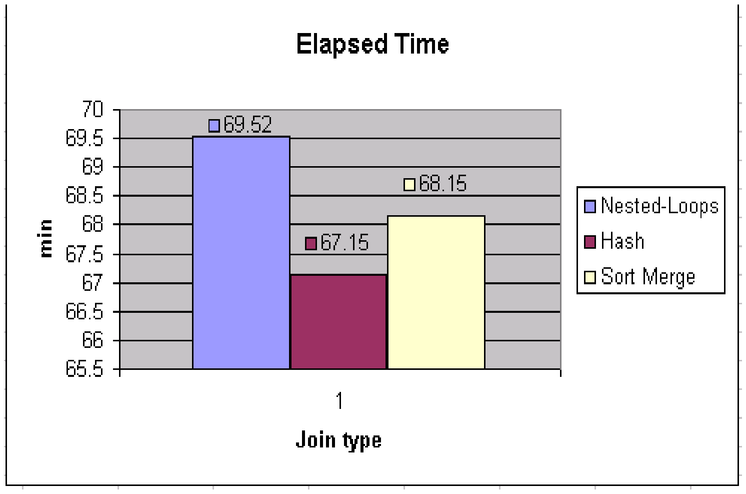


Fig 1.5 - The response time *with fetching data* for all the algorithms, in the second case.

Another interesting “performance time” analysis should be the execution time of queries for each type of *join* in both cases, without fetching data in the grid. After studying this problem the results relative to the execution time are really amazing (see Figs. 1.6 and 1.7), compared to those relative to the execution time with fetching data (previously obtained - see above). To display the results for each case we'll add to the join condition any number that is not integer, e.g. in our case for *use_hash* we'll use the value *+0.1*.

To view the execution time the following syntax is used:

```
SELECT /*+ USE_HASH (a b) */
    name_soft, version, size_soft, name_vendor
FROM software_table a, vendor_table b
WHERE a.vendor_id = b.vendor_id+0.1;
```

In the first case, the results for each join algorithm are:

USE_NL= 609 seconds
USE_HASH: 0.328 seconds
USE_MERGE: 2 seconds

In the second case, we have the following times:

USE_NL= 4200 seconds
USE_HASH: 10 seconds
USE_MERGE: 45 seconds

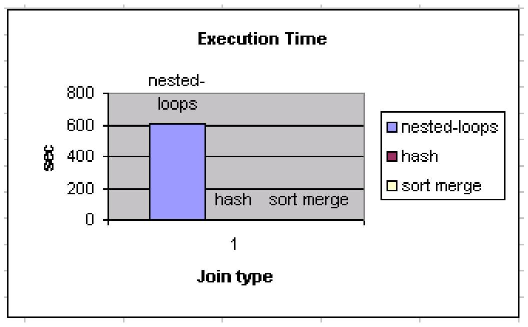


Fig. 1.6 - The response time *without fetching data* (the first case).

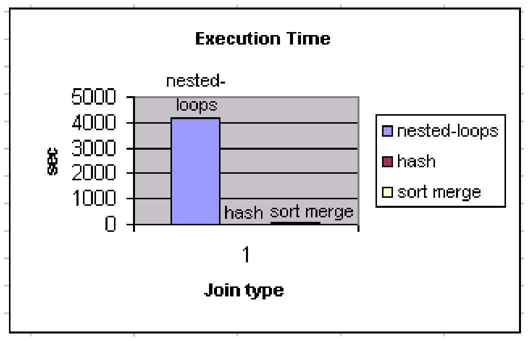


Fig. 1.7 - The response time *without fetching data* (the second case).

Conclusions

The leak of practical examples requested discussions around this subject. The evaluation tests based upon 'execution cost' that this article proposed are indeed an original and useful approach of the matter. These tests, along with the 'optimization plan', raised prevalent practical conclusions.

As the study presented in this article reveals, the number of recordings plays a very important role in choosing the type of algorithm. More records in a table means an exponential increasing of the execution time, no matter the type of algorithm in use.

This method is also based upon the cost of execution (CBO). If more parameters are taken into account, as the plan of execution has already shown (db block gets, memory, disk, CPU etc.), the optimizer makes a compromise between the interval of execution and the minimum of resources. In both cases, the *hash join* algorithm performs faster than the other nested-loops algorithms, as shown by the graphics above. The graphics also point differences between the interval of execution 'without fetch' and the interval of execution 'with fetch'. In these two cases, the *hash algorithm* is the fastest, followed by the *merge algorithm*, the *nested-loops algorithm* being the slowest. We notice that in the case of nested-loops algorithm, the intervals of execution with or without data loading are very much alike, as the data loading occurs during the execution time.

There is no algorithm faster than other and the optimizer only chooses one of them, depending on the practical situation.

This article presented technical information regarding the optimization methods using *JOIN sql*, an *Oracle* specific, based on original test cases and situations, which are not well documented and that can either be discovered only by experienced database users or obviously known by the database designers.

REFERENCES

1. *Eyal Aronoff, Kevin Loney and Noorali Sonawalla*, Explain Plan: Everything you wanted to know and had no-one to ask.
2. *Joseph C. Johnson*, Oracle9i™ Performance Tuning, Study Guide, 2002
3. *Dominique Jeunot*, Enterprise DBA Part 2:Performance and Tuning, **vol. 1**, Instructor Guide, September, 1999
4. *** Database Performance Tuning Guide and Reference in Oracle9i, October, 2002
5. http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/admin/c0005311.htm
6. http://www.psoug.org/reference/hints.html
7. http://oracle.developpez.com/guide/tuning/tkprof/
8. http://www.adp-gmbh.ch/ora/sql/hints.html
9. http://en.wikipedia.org/wiki/Join_%28SQL%29

10. http://www.cs.umbc.edu/help/oracle8/server.815/a67781/c20c_joi.htm#4479
11. http://people.aapt.net.au/roxsco/tuning/hints.html
12. http://www.lc.leidenuniv.nl/awcourse/oracle/server.920/a96533/optimops.htm
13. http://www.lc.leidenuniv.nl/awcourse/oracle/server.920/a96533/optimops.htm#49941
14. http://www.lc.leidenuniv.nl/awcourse/oracle/server.920/a96533/toc.htm