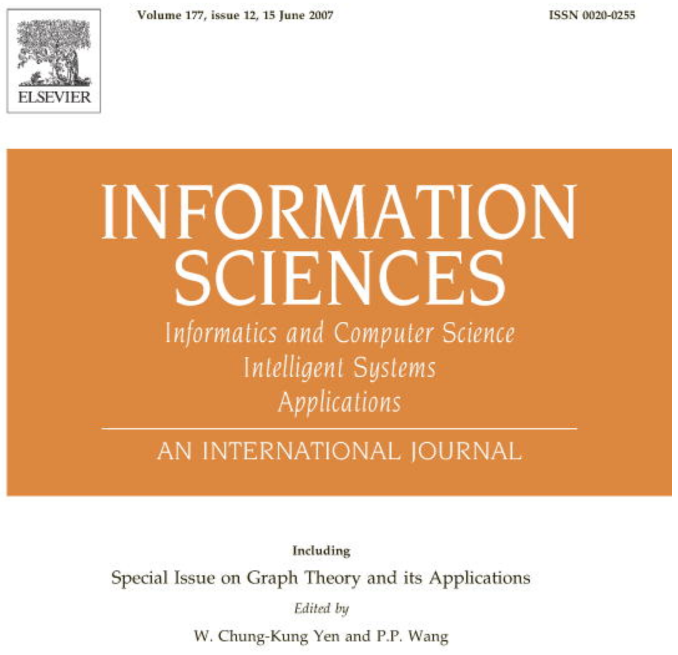


Volume 177, issue 12, 15 June 2007

ISSN 0020-0255

INFORMATION SCIENCES

Informatics and Computer Science
Intelligent Systems
Applications

AN INTERNATIONAL JOURNAL

Including
Special Issue on Graph Theory and its Applications
Edited by
W. Chung-Kung Yen and P.P. Wang

Available online at www.sciencedirect.com
ScienceDirect
Information Sciences 177 (2007) 2493–2521
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

The use of Hints in SQL-Nested query optimization

David Taniar [a,*], Hui Yee Khaw [a], Haorianto Cokrowijoyo Tjioe [a], Eric Pardede [b]

[a] *Clayton School of Information Technology, Monash University Clayton, Vic. 3800, Australia*
[b] *Department of Computer Science & Computer Engineering, La Trobe University, Bundoora, Vic. 3083, Australia*

Received 17 April 2004; received in revised form 3 April 2006; accepted 13 December 2006

Abstract

The query optimization phase in query processing plays a crucial role in choosing the most efficient strategy for executing a query. In this paper, we study an optimization technique for SQL-Nested queries using Hints. Hints are additional comments that are inserted into an SQL statement for the purpose of instructing the optimizer to perform the specified operations. We utilize various Hints including *Optimizer Hints*, *Table join and anti-join Hints*, and *Access method Hints*. We analyse the performance of various nested queries using the TRACE and TKPROF utilities which provide query execution statistics and execution plans.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Query optimization; SQL; Nested query; Hints; TRACE; TKPROF

1. Introduction

Query optimization is the most important stage in query processing where the database optimizer has to choose a query-evaluation plan with minimized cost and maximized performance [4,13,25,27]. The functionality of query optimization in deciding the best query execution plan is a significant task since there are a lot of possible queries in terms of the canonically equivalent algebraic expression for one given SQL query [4,7,8]. Moreover, the response time for those algebraic representations can vary one from another. Nonetheless, the efficiency of the chosen plan could make a great difference to the system's performance and hence this step should not be neglected. This is especially the case when the query being optimized consumes a considerable amount of resources such as physical disk reads and memory buffer. Thus, reasons such as these have made the task for query optimization critically important.

There are three essential components of a query optimizer [4,15]: (1) search space, (2) cost modeling, and (3) search strategy. In the search space, there are several possible canonical equivalent query expressions whose

* Corresponding author. Tel.: +61 3 9905 9693; fax: +61 3 9905 5159.
E-mail addresses: David.Taniar@infotech.monash.edu.au (D. Taniar), Haorianto.Tjioe@infotech.monash.edu.au (H.C. Tjioe), E.Pardede@latrobe.edu.au (E. Pardede).

0020-0255/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2006.12.015

function is not merely to reduce the cost. It is the job of the cost modeling component to estimate the cost of those query expressions based on the database's statistical information. The search strategy will choose the most efficient execution plan for a given query task by investigating the search space.

Generally, in the query optimization process, the system will automatically choose the most efficient optimization technique [15,27]. There are two commonly used query optimization techniques: *Heuristics* and *Cost estimation* optimization [4,10]. For example, in a commercial database industry, Oracle adopts these most common optimization techniques [16]. By applying *Heuristics* optimization, the algebraic representation uses a set of Heuristics rules to transform the query which selects the appropriate operator trees in a query search strategy in order to give the efficient query execution plan [18]. However, one should not only depend on the *Heuristics* technique; another consideration should be the cost involved in executing the query by applying different search strategies and selecting the cheapest cost estimation. The *Cost estimation* technique predicts the cost of different execution strategies and selects the cheapest execution cost [5,13,17]. Unfortunately, these optimization techniques are not enough to serve the best query optimization technique. The *Heuristics* technique, although it is relatively stable when compared with the cost estimation technique, still does not serve the most efficient optimization plan [17]. The *Cost estimation* technique, although it serves the most efficient optimization technique based on the cost estimation model, does not always give the best optimization technique and there are many search strategy options which are too time-consuming when estimating these search strategies [3,4].

Moreover, since the Database Administrator (DBA) often has knowledge about their data better than the system optimizer, the application of *Hints* as the optimization technique on the SQL statement will give the DBA the ability to manually turn on the SQL statement in order to improve system performance with less resource consumption compared with the *Heuristics* and *Cost estimation* techniques. Last but not the least, the *Hints* optimization technique is capable of choosing the best query transformation based on the user's specified comment which is added to the SQL query statement [23].

Basically, *Hints* are some additional comments that are placed within an SQL statement aiming to directly guide the current system optimizer to alter the optimized execution plan. It can directly decide the execution plan, guided by a user's specified notes inserted within an SQL statement [2]. *Hints* can be categorized into *optimizer Hints*, *table join and anti-join Hints*, *access method Hints* [2,16].

In this paper, we introduce nested query optimization using Hints in the context of relational database management systems (RDBMS), since the modern database applications usually deal with many complex queries in RDBMS and those complex queries often involve nested queries. Moreover, a nested query optimization consumes a larger number of resources such as physical disk reads and memory buffers when compared with a non-nested query [25]. Hints could significantly speed up the system's performance and reduce the resources needed for processing the nested queries. Here, we particularly focus on standard *nested query*, *non-equality nested query*, *semi-join nested query* and *anti-join nested query*.

There is work on the use of Hints in SQL for certain system such as [1] for Sybase. In this paper, our implementation will be done using Oracle. In Oracle itself, the use of Hints for Object-Relational queries has been published in our previous paper [23].

The organization of the rest of the paper is as follows. We discuss the related works in Section 2. In Section 3 we will introduce Hints and its specification. In Section 4, we discuss a nested query optimization framework; this is followed by performance evaluation with some results showing the effectiveness of using Hints in Section 5. Finally in Section 6, we give our conclusions and discuss future work.

2. Related works

Kim [14] first introduced an efficient technique to improve the processing of nested queries. The idea is to rewrite the initial nested query form to its canonical equivalent non-nested form. The traditional technique in implementing nested queries can be very inefficient since, by using the nested-iteration method, it has to retrieve the inner query block once of each tuple of its outer query block. This is a very time-consuming process. First, he identified five types of nested queries which are: *type-A nesting*, *type-N nesting*, *type-J nesting*, *type-JA nesting* and *type-D nesting*. As can be seen in Table 1, there are certain conditions which stated the

Table 1
Kim's nested queries classification

Conditions	Type A	Type N	Type J	Type JA	Type D
Inner query block Q has a join predicate that references the outer query block	No	No	Yes	Yes	Yes
SELECT clause of Q includes an aggregate function	Yes	No	No	Yes	No
Inner query block$Q$ contains a division predicate	No	No	No	No	Yes

criteria of each nested query type. The first condition is that the inner query block Q has a join predicate that references the outer query block. The second condition is that the SELECT clause of Q includes an aggregate function. The final condition is that the inner query block Q contains a division predicate.

Based on these classifications, Kim [14] proposed sets of algorithms to transform the initial nested queries to their canonical equivalent non-nested form. This allows the query optimizer to choose a merge join algorithm when implementing nested queries rather than applying the high cost nested-iteration method. However, Ganski and Wong [7] later discovered some deficiencies in Kim's algorithms [14] and have attempted to fix these bugs. Nonetheless, Kim's idea of query rewriting became the foundation of today's evolution of query optimizer approaches [14].

There are two commonly used query optimization techniques which are: the *Heuristics* and the *Cost estimation* technique. A *Heuristics* method transforms the initial query presentation into an efficient query tree equivalent based on the order of operation for executing a query [6,18]. For example, in the Volcano [10] and the Cascades [9] systems, the ordering of operations for executing a query is applied widely to represent the knowledge of search space [4]. The main idea of this technique is to change the internal representation of a query tree structure by using the Heuristics technique. Generally, the SELECT and PROJECT operations, which reduce the number of records and the number of attributes respectively, are found to be the most restrictive operations [6]. So there could be a great reduction in resource usage if some of the restrictive operations, which can efficiently reduce the intermediate result size, are executed first.

In addition to the *Heuristics* technique, the *Cost estimation* technique seems to be more effective for many nested queries operations [5,11,17]. The system optimizer needs information about database statistics that is kept in the DBMS's catalogue in order to be able to estimate the costs of various execution plans generated [5]. For example, MAGIC [17] is one of methods which apply the *Cost estimation* technique. It uses the rewriting method as a special join method to be added into any existing *Cost estimation* query optimizer. Although the *Cost estimation* technique is capable of providing reasonably accurate estimates, it should be borne in mind that these are only *estimates* and hence, the chosen execution plan might not be the cheapest [4]. After some alternative execution plans have been generated, the *Cost estimation* technique optimizer uses the statistics to predict the cost of each plan and chooses the one with the lowest cost.

Due to the estimation nature of this technique, the need for Hints has emerged. The introduction of Hints for nested query optimization has indicated the inability of those commonly used optimization techniques to produce the best strategy for every nested query. By applying *Hints* as an optimization technique, the search strategy is directed to choose the optimize execution plan based on the specified knowledge which is stated in *Hints* [2,16]. There are two main goals of Hints optimization: a faster response time, and lower resource consumption. Moreover, the Hints approach is easy to understand in terms of human language and its process is simple to follow.

3. Hints: a background and classifications

As can be seen in Fig. 1, all Hints have to be written after a SELECT /UPDATE /INSERT/ DELETE statement with `/*+ Hint */`. In this example, we use *USE_HASH* Hint to optimize the nested query between cust_order and staff tables. Oracle provides an extensive list of Hints that can be used to tune various aspects of a query performance [16].

We categorize Hints for nested query optimization into three parts: (1) optimizer Hints, (2) table join and anti-join Hints, (3) access method Hints. Detailed explanations of these are given in the following sections.

```
SELECT * FROM Cust_order r                        a HINT statement
WHERE r.staffid IN
              (SELECT /*+ USE_HASH(s, r)*/ s.staffid FROM staff s)
```

Fig. 1. Using a Hint in an SQL statement.

3.1. Optimizer Hints

Optimizer Hints apply a different optimizer to a query, which in turn redirects the overall processing goal [1]. Oracle offers four kinds of optimizer Hints, which are *all_rows*, *first_rows*, *rule*, and *choose* (refer to Fig. 2). The *all_rows* Hint explicitly chooses the cost estimation technique to optimize a statement block with a goal of best throughput (that is, minimum total resource consumption).

The *choose* Hint causes the optimizer to choose between the Heuristics optimization and cost estimation techniques for an SQL statement. The optimizer bases its selection on the presence of statistics for the tables accessed by the statement. If the data dictionary has statistics for at least one of these tables, then the optimizer uses the cost estimation technique and optimizes with the goal of best throughput. If the data dictionary does not have statistics for these tables, it uses the Heuristics optimization technique.

The *first_rows* Hint favors a full-index scan and invokes the cost estimation technique to return the first row of a query within the shortest processing time and with minimum resource consumption. Since its goal is to achieve the best response time, it is suitable for use in Online Transaction Processing (OLTP).

Fig. 3 shows that the optimizer goal has been changed from No-Hint to add *first_rows* Hint to the select statement. In Fig. 3a the goal of select statement is rule (Heuristics technique) in which we do not apply any Hint since it is a default optimization technique from the system. By applying *first_rows* Hint in Fig. 3b, the select statement goal is forced to change from the system default optimization. Although the execution plan between Figs. 3a and b is quite the same, however by applying a Hint, it gives a better performance result as can be seen in our performance evaluation in Section 5.

3.2. Table join and anti-join Hints

There are several Hints in the table join and anti-join Hints provided by Oracle as shown in Fig. 4, which are: *use_nl* Hint, *use_merge* Hint and *use_hash* Hint. The *use_nl* Hint causes Oracle to join each specified table to other row sources with a nested loop join that uses the specified table as the inner table. In general, the default join method in Oracle is the nested loop join [24]. Therefore, *use_nl* Hint is not used in the performance evaluation. There are six Hints which can be used for join nested query optimization:

- The *use_hash* Hint. The Hint invokes a hash join algorithm to merge the rows of the specified tables. In many circumstances, it changes the access path to a table in addition to the change in join method [20].
- The *use_merge* Hint. Alternatively, the Hint forces the optimizer to join tables using the sort–merge operation [21].

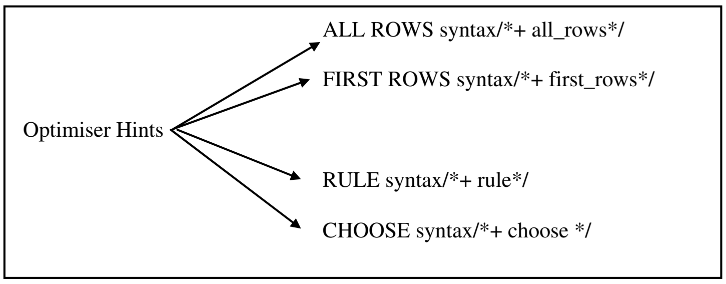
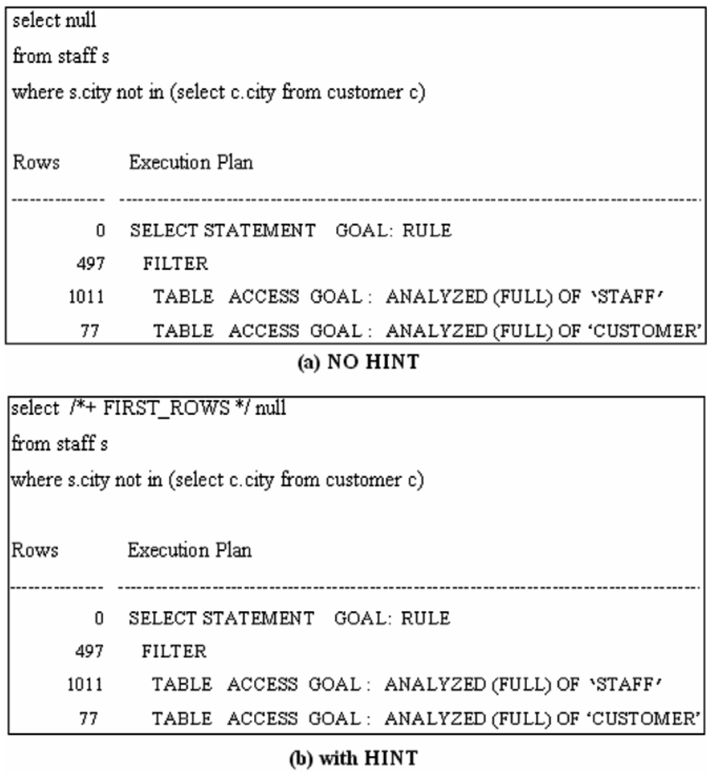

Optimiser Hints
ALL ROWS syntax/*+ all_rows*/
FIRST ROWS syntax/*+ first_rows*/
RULE syntax/*+ rule*/
CHOOSE syntax/*+ choose */

Fig. 2. Optimizer Hints.



```
select null
from staff s
where s.city not in (select c.city from customer c)

Rows          Execution Plan
-------------- ---------------------------------------------------------------------------------
         0   SELECT STATEMENT   GOAL: RULE
       497     FILTER
      1011       TABLE  ACCESS  GOAL :  ANALYZED (FULL) OF 'STAFF'
        77       TABLE  ACCESS  GOAL :  ANALYZED (FULL) OF 'CUSTOMER'
```

(a) NO HINT

```
select  /*+ FIRST_ROWS */ null
from staff s
where s.city not in (select c.city from customer c)

Rows          Execution Plan
-------------- ---------------------------------------------------------------------------------
         0   SELECT STATEMENT   GOAL: RULE
       497     FILTER
      1011       TABLE  ACCESS  GOAL :  ANALYZED (FULL) OF 'STAFF'
        77       TABLE  ACCESS  GOAL :  ANALYZED (FULL) OF 'CUSTOMER'
```

(b) with HINT

Fig. 3. The effect of adding an optimizer Hint.

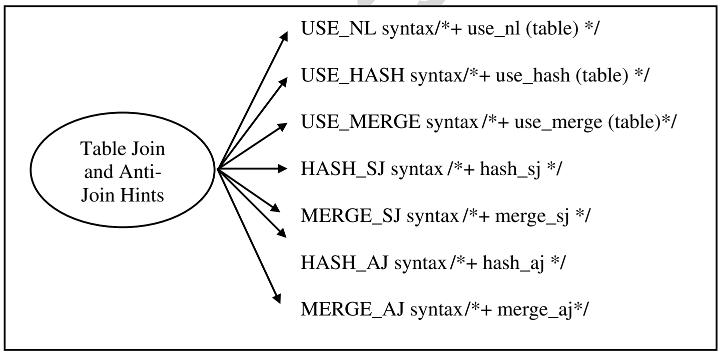


Table Join and Anti-Join Hints
USE_NL syntax/*+ use_nl (table) */
USE_HASH syntax/*+ use_hash (table) */
USE_MERGE syntax /*+ use_merge (table)*/
HASH_SJ syntax /*+ hash_sj */
MERGE_SJ syntax /*+ merge_sj */
HASH_AJ syntax /*+ hash_aj */
MERGE_AJ syntax/*+ merge_aj*/

Fig. 4. Table join and anti-join Hints.

- The *hash_sj* Hint. This is a semi-join Hint that is added to the inner block of a correlated nested query using the EXISTS clause for the purpose of transforming the query into a hash semi-join to access the specified table.
- The *merge_sj* Hint. Similarly, the Hint also transforms a correlated EXISTS nested query into a semi-join except that it invokes a merge semi-join.
- The *hash_aj* Hint. The Hint is placed in the inner loop of a NOT IN nested query to invoke a hash anti-join operation.
- The *merge_aj* Hint. The Hint joins the tables in a NOT IN nested query using the merge anti-join operation.

As can be seen from Fig. 5a, a *use_merge* Hint has two parameters which are *S* for table *Staff* and *C* for table Customer to be joined to the row source resulting from joining the previous tables in the join order using a sort–merge join. Similar to Fig. 5a, in Fig. 5b a *use_hash* Hint has been specified two parameters which are *S* for table *Staff* and *R* for table *Cust_order* to be joined to the row source resulting from joining the previous tables in the join order either using a hash join.

```
SELECT * FROM Staff S
WHERE S.City in
(SELECT /*+ USE_MERGE(S,C) */ C.City
From Customer C )
```

(a) USE_MERGE hint

```
SELECT * FROM Staff S
WHERE S.Staffid > ANY
(SELECT /*+ USE_HASH(S,R) */
R.Staffid FROM CUST_ORDER R)
```

(b) USE_HASH hint

Fig. 5. Table join and anti-join Hints examples.

```
SELECT /*+ INDEX(V,Inventory_invid_pk) */ *
FROM Inventory V
WHERE V.Curr_price in
      (SELECT O.Order_price
        FROM Orderline O
        WHERE V.Invid = O.Invid)
```

Fig. 6. An example of access method Hints.

3.3. Access method Hints

The access method Hints can be classified into *Hints without index* and *Hints with index*. *Rowid* Hint is one of *Hints without index* with syntax: `/*+ Hint(table) */`. *Hints with index* consist of *index* Hint, *index_join* Hint and *index_ffs* Hint with syntax: `/*+ Hint (table, index) */`. As shown on Fig. 6, there is a SELECT statement with *index* Hint consist of two parameters which are *V* as the alias of table *Inventory* and *Inventory_invid_pk* as the table index of table *Inventory*. We will briefly explain each Hint in the access method Hints as follows:

- The *Rowid* Hint. The Hint forces the optimizer to scan the specified table by row id, which is generally gathered from an index.
- The *Index* Hint. The Hint explicitly instructs the optimizer to use an index scan as the access path to the specified table.
- The *Index_join* Hint. The Hint alerts the optimizer to access the specified table using the index join method [22].
- The *Index_ffs* Hint. The invoked index fast full scan operation by the Hint will fully access an index in random order. It is suitable to be used in cases where access to index alone is sufficient to resolve the query. In other words, it is not necessary to access the table rows.

4. Nested query optimization framework

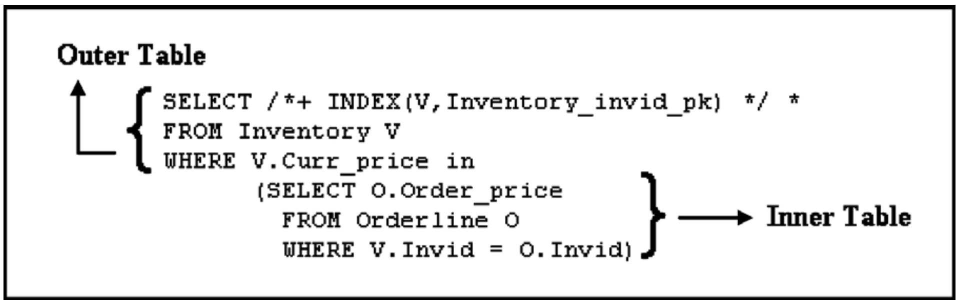
In this section, we explain the nested query and the optimizer modes used in our experimentations.

4.1. Nested queries

Basically, a nested query is a query within another query in a nested form and can be classified into four groups [6,7,13], which are:

- Standard nested queries (IN),
- Non-equality nested queries (ANY/ALL with conditional operators),
- Semi-join nested queries (EXIST),
- Anti-join nested queries (NOT EXIST and NOT IN).

However, we categorize the nested query taxonomy according to the clauses used, which include EXIST, IN, ANY/ALL, NOT EXISTS, and NOT IN. Therefore, there will be five major types of nested queries where each group is further divided into various join structures.



```
SELECT /*+ INDEX(V,Inventory_invid_pk) */ *
FROM Inventory V
WHERE V.Curr_price in
      (SELECT O.Order_price
        FROM Orderline O
        WHERE V.Invid = O.Invid)
```

Fig. 7. Inner and outer table.

The first level of division within each defined clause is related to the correlation between the inner table and the outer table in a nested query. According to Kim [14], a correlated nested query is where a join predicate of the inner table references the outer table (see Fig. 7). Thus, for every outer row processed, the inner table has to be evaluated once. If the correlated outer table is large, the processing could cause an extensive scan of the inner table. On the other hand, a non-correlated nested query has to be executed once only in order to produce a temporary output, which will serve as an input to the outer table for further processing. The recursive processing on the inner table is eliminated, as it does not reference the outer table. In the second level of division, the classification of join structure is based on the characteristics of indexed attribute and non-indexed attribute. In the case of nested queries, a primary table can be either in the inner or the outer block query. Therefore, the two scenarios will be considered in the experiments in order to analyze the effect of index on both the inner and the outer blocks.

The IN clause is one of the most commonly used operators in standard nested queries. It returns a row from the outer table if the value in the attribute before the clause is equal to one value, or a set of values, produced by the inner query. The processing is similar to the equi-join where the values of two tables are compared on a common attribute. In correlated nested queries, further classification is made on the position of the primary table (see Fig. 8(a)), while the characteristics of attributes required by the IN clause categorize non-correlated nested queries into classes (see Fig. 8(b)).

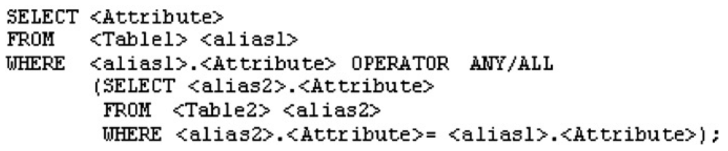
The use of the ANY or ALL clause in correlated Non-equality nested queries is rather complex. Therefore, any structure where the correlation is between non-key attributes will not be considered. Moreover, it is not practical to implement the structure in the real world due to its futility. Fig. 9(a) shows the syntax of using correlated non-equality nested queries while Fig. 9(b) shows the syntax of using non-correlated non-equality nested queries. Nonetheless, the ANY and the ALL clauses have to be used in sync with conditional operators (see Figs. 9(a) and 9(b)).

```
SELECT <Attribute>
FROM   <Table1> <alias1>
WHERE  <alias1>.<Attribute> IN
   (SELECT    <Attribute>
     FROM      <Table2> <alias2>
     WHERE     <alias2>.<Attribute> = <alias1>.<Attribute>);
```

Fig. 8(a). Correlated IN clause nested queries.

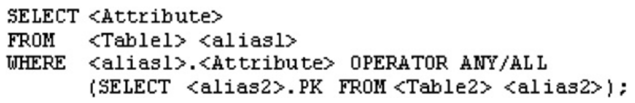
```
SELECT <Attribute>
FROM   <Table1> <alias1>
WHERE  <alias1>.<Attribute> IN
       (SELECT <alias2>.PK FROM  <Table2> <alias2>);
```

Fig. 8(b). Non-correlated IN Clause nested queries.



```
SELECT <Attribute>
FROM   <Table1> <alias1>
WHERE  <alias1>.<Attribute> OPERATOR  ANY/ALL
       (SELECT <alias2>.<Attribute>
        FROM  <Table2> <alias2>
        WHERE <alias2>.<Attribute>= <alias1>.<Attribute>);
```

Fig. 9(a). Correlated non-equality nested queries.



```
SELECT <Attribute>
FROM   <Table1> <alias1>
WHERE  <alias1>.<Attribute> OPERATOR ANY/ALL
       (SELECT  <alias2>.PK  FROM <Table2> <alias2>);
```

Fig. 9(b). Non-correlated non-equality nested queries.

A nested query that uses the EXISTS clause is also called a *semi-join nested query*. This is because the matching row in the outer table would be displayed once only even if there are multiple rows produced by the inner query [2]. Besides, Burleson [2]) proves that it is inappropriate for non-correlated nested queries to use the EXISTS clause, as the result produced would be incorrect. Therefore, the taxonomy will include only the correlated nested queries in the experiments. The following illustration shows the structure of a *correlated semi-join nested query.*

```
SELECT        ⟨Attribute⟩
FROM          ⟨Table1⟩ ⟨alias1⟩
WHERE         EXISTS
     (SELECT       ⟨Attribute⟩
     FROM          ⟨Table2⟩ ⟨alias2⟩
     WHERE         ⟨alias2⟩.⟨Attribute⟩=⟨alias1⟩.⟨Attribute⟩);
```

An *Anti-join nested query* using the NOT EXISTS or NOT IN clause is similar to that using the EXISTS clause or IN clause. Yet, in the former case, a row in the outer table would be retrieved only if the condition specified in the inner query does not exist. In other words, the final result excludes the matching rows in the inner query. The following illustration shows the structure of an *anti-nested query.*

```
SELECT        ⟨Attribute⟩
FROM          ⟨Table1⟩ ⟨alias1⟩
WHERE         NOT EXISTS/NOT IN
              (SELECT   ⟨Attribute⟩
              FROM     ⟨Table2⟩ ⟨alias2⟩
              WHERE     ⟨alias2⟩.⟨Attribute⟩ = ⟨alias1⟩.⟨Attribute⟩);
```

4.2. Optimizer modes

There are two kinds of optimizer modes in Oracle: the *Heuristics* optimizer and the *Cost estimation* optimizer [16]. The *Heuristics* optimizer's main function is to reduce the size of the intermediate result. In general, the SELECT and PROJECT operations, which reduce the number of records and the number of attributes respectively, are found to be the most restrictive operations [6]. Apparently, the re-ordering of the leaf nodes on the query tree can have a significant impact on the system's performance and, hence, good Heuristics should be applied whenever possible. However, as shown in the performance evaluation section, this optimizer mode is less effective on many join nested queries optimizations when compared with using Hints on CBO optimizer mode.

The *Cost estimation* optimizer determines which execution plan is the most efficient by considering available access paths and by factoring in information based on statistics for the schema objects (tables or indexes) accessed by the SQL statement [11]. For instance, the decision to choose between an index scan and a full-table scan for a table with skewed (disproportional) data will not always be obvious, and hence cannot be concluded without first examining the statistics.

A *Cost estimation* optimizer will favor an index scan over a full-table scan if a given query retrieves less than 40% of the data in a table. On the other hand, a full-table scan will be preferred if the retrieved data will be over 60%. As can be seen, the *Cost estimation* optimizer is more flexible and intelligent in choosing the SQL operations. Therefore, it should make a better execution plan than the *Heuristics* optimizer does if it has all the statistical information it needs. Moreover, *Cost estimation* has two main optimization goals: the fastest response time with minimum resource usage, and the best throughput with minimum total resource consumption.

5. Performance evaluation

In this section, we present our performance evaluation result incorporating Hints using TKPROF utility in Oracle.

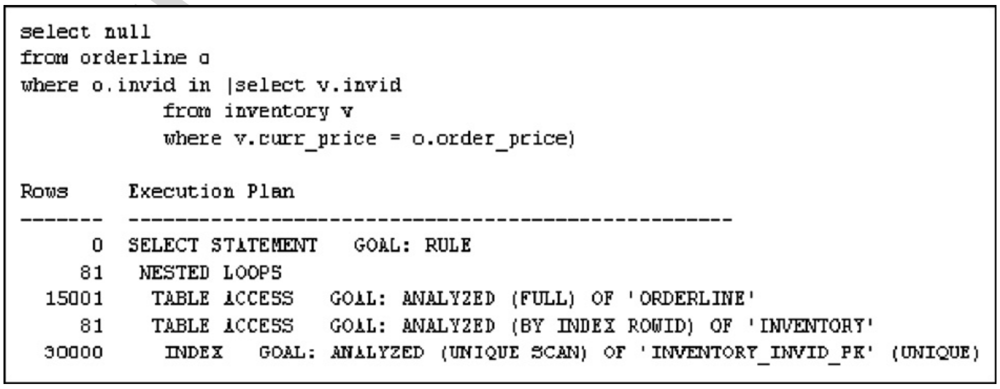
5.1. Execution plan (query tree) and TKPROF utilities

In general, an execution plan is the strategy produced and used by the optimizer to process a query in order to get the desired result [12]. It is also known as the query tree due to its tree-like structure. It is essential to understand the functioning of the execution plan, as it provides a useful indication of the performance of the query processing. For instance, as shown in the following illustration (Fig. 10), an indexed nested loop join is used to join the tables in the given query. The join method may indicate an efficient processing, as the previous section explained that the presence of an index in the inner table may dramatically improve the performance of the nested loop join.

However, the operations used in the execution plan are only an *indicator,* not an absolute answer. Moreover, its main function is to explain the steps involved in the processing and the sequence of execution. As can be seen, there are four steps involved in the processing of the given query:

Step 1: Nested loops;
Step 2: Full-table access to *Orderline* table;
Step 3: Table access to *Inventory* table by index ROWID;
Step 4: Index unique scan on the primary key of the *Inventory* table.

The operations below the nested loops are indented indicating that they are executed prior to the join operation. Hence, the first execution is Step 2, which will retrieve a row from the *Orderline* table and return it to



```
select null
from orderline o
where o.invid in (select v.invid
            from inventory v
            where v.curr_price = o.order_price)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
     81   NESTED LOOPS
  15001    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'ORDERLINE'
     81    TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'INVENTORY'
  30000     INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'INVENTORY_INVID_PK' (UNIQUE)
```

Fig. 10. Example of execution plan.

Step 1. Then, Step 1 will send the *invid* (inventory number) to Step 4 for every row obtained from Step 2. In Step 4, the optimizer will perform an index unique scan on the *invid* to gather the ROWID. If no ROWID is found, Step 3 will instruct Step 1 to discard the row. Otherwise, Step 3 will use the ROWID to access the *Inventory* table to retrieve the corresponding row. Finally, Step 3 will return the row to Step 1, where the row will be joined with that from Step 2 if the current price is equal to the order price. If the two prices are not the same, Step 3 will not return a row, and Step 1 will eliminate the row from the result set. The steps will be carried out recursively until all the rows have been processed.

The TRACE utility is also known as the SQL trace facility, which can be activated only by setting the relevant parameters. There are two types of parameters that need to be carefully set [2]: a file size and location parameters (eq. User_dump_dest = c:\orahome\tracefile) b function enabling parameters (e.g. time_statistics and sql_trace). However, the report produced is difficult to understand. Therefore, it must be used in conjunction with the TKPROF utility, whose task is to translate the traced result into a readable format [2]. When the Oracle database is installed, TKPROF is automatically installed in the directory \orahome\bin. Firstly, we add \orahome\bin to the current path. Then, the utility is invoked by using the following command:

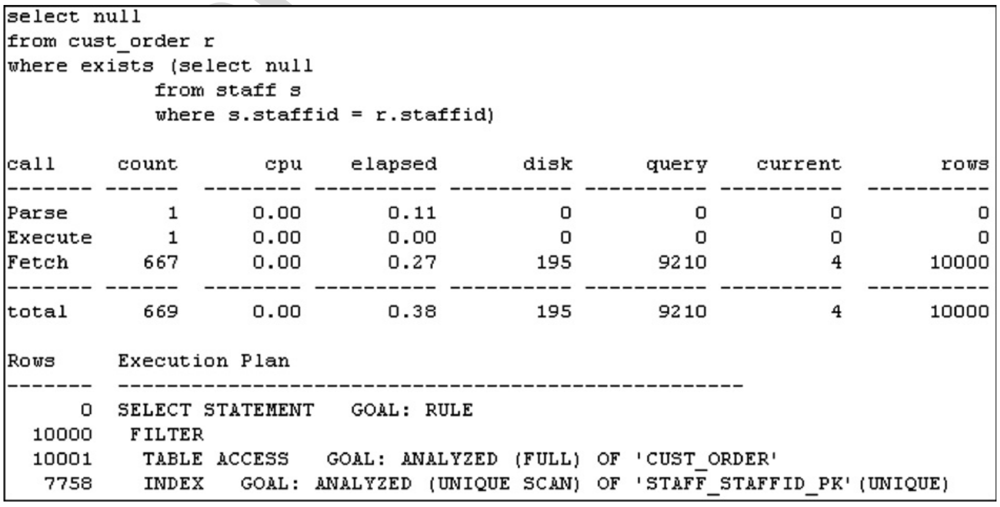
$$\text{Tkprof}\langle\text{TraceFile}\rangle\langle\text{OutputFile}\rangle[\text{explain} = \langle\text{username}\rangle/\langle\text{password}\rangle]$$

The trace file has an extension of *.trc*, while the output file that is in the readable format will have an extension of *.prf*. It is essential to specify the location in both the ⟨TraceFile ⟩and the ⟨OutputFile ⟩, otherwise, TKPROF would not be able to perform its task due to the missing file. The following shows the command line we use to invoke TKPROF: `Tkprof d:\ora0001.trc d:\trace.prf explain = system/manager`.

As shown in Fig. 11, the TKPROF result is divided into three parts which are: *sql statement*, *statistic information* and *execution plan*. The statistics section contains seven statistics measurements and one call column (the first column) indicating the phases of query execution. The following will provide a description of the phases and the statistics measurements.

5.1.1. Phases

- *Parse*. The parser checks the syntax and the semantics of the SQL statement, which will be decomposed into relational algebra expressions for execution.
- *Execute*. After the parsing and validation stage, the optimizer will execute the decomposed SQL statement using the execution plan operations shown in Fig. 11.
- *Fetch*. Finally, in the fetching phase, the optimizer will retrieve the final output from the corresponding tables based on the instruction passed from the query execution phase.



```
select null
from cust_order r
where exists (select null
             from staff s
             where s.staffid = r.staffid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.11          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      667      0.00       0.27        195       9210          4       10000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      669      0.00       0.38        195       9210          4       10000

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
  10000   FILTER
  10001    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST_ORDER'
   7758    INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'STAFF_STAFFID_PK'(UNIQUE)
```

Fig. 11. Example of TKPROF result (formatted trace report).

5.1.2. Statistics

- *Count*. The number of times the phases (parse, execute, fetch) are called.
- *CPU*. The total CPU time (in hundredths of a second) taken for each phase to perform.
- *Elapsed*. The total processing time for each phase to complete their tasks.
- *Disk*. The total number of physical disk reads.
- *Query*. The total number of data buffers retrieved from memory, which is generally used by the SELECT statement.
- *Current*. The statistic is similar to the query statistics, which also indicates the total number of data buffers retrieved from memory except that this mode is generally used by the UPDATE, INSERT, and DELETE statements.
- *Rows*. The total number of rows processed by the SQL statement.

5.2. Experimental results

For the purpose of experimentation in the paper, a Sales database has been created as a sample database. Table 2 shows the details of the database, while Fig. 12 depicts the relationships between each individual table. We perform the experiments on Pentium IV 1.8 GHz CPU with 256MB. We use Oracle9i Database to perform our SQL queries. The Exception is given to table Cust and table Staff; in our experimental results the terms Cust and Customer are interchangeable to identify table Cust; also in our experimental results we use name Staff_S and Staff interchangeably to identify table Staff.

In the next section, we use an experimental *Hints* optimization technique and also an optimization technique without using Hints. For the No-Hint approach, the optimizer system will choose the commonly used optimization techniques which are: the *Heuristics* and *Cost estimation* techniques. We also compare those optimization techniques results based on the join nested query framework in Section 4. These nested queries are

Table 2
Sales database

Table	Attributes	Records
Cust	(***custid***, cname, addr, city, pcode)	10,010
Staff	(***staffid***, sname, city, pcode)	1010
Cust_order	(***ordered***, orderdate, custid, staffid)	10,000
Item	(***itemid***, itemdesc, category)	999
Inventory	(***invid***, itemid, curr_price, qoh)	9500
Orderline	(***ordered***, ***invid***, order_price, qty)	15,000

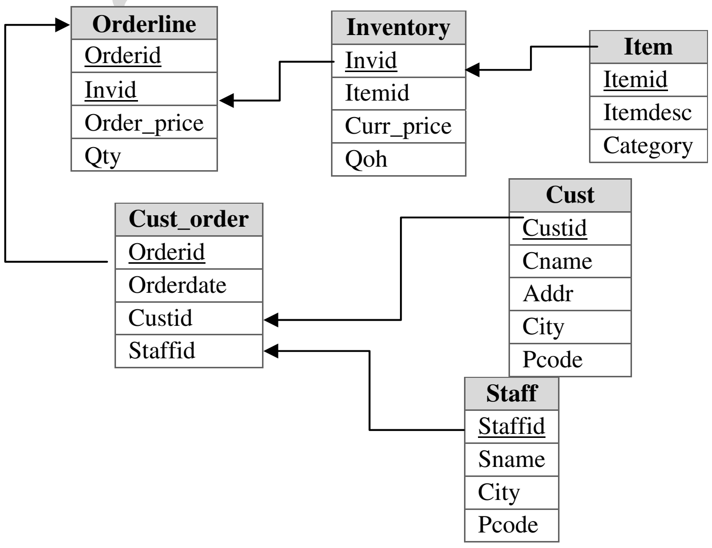


Fig. 12. Sales order systems.

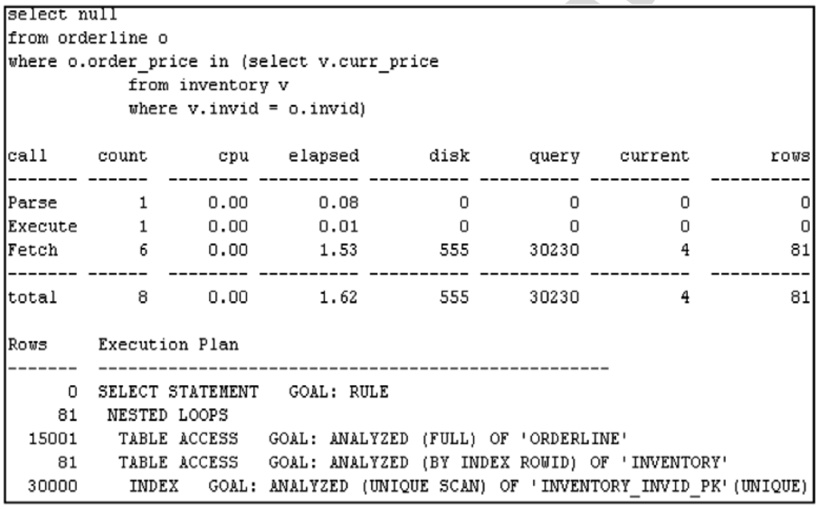
IN clause nested queries, EXISTS clause nested queries; ANY/ALL clause nested queries and NOT EXISTS and NOT IN clause nested queries. Nested queries with aggregate functions are not within the experimentation scope of this paper.

In the following comparisons as we will see in the next section, the place to add a Hint (inner or outer loop) is specified by enclosing an 'O' or 'I' at the end of the Hint indicating the outer loop and the inner loop respectively. For example, *use_hash* (O) means that the Hint is added to the outer loop of a nested query. Notice that, the '(O)' is not part of the Hint.

5.2.1. IN clause nested queries results

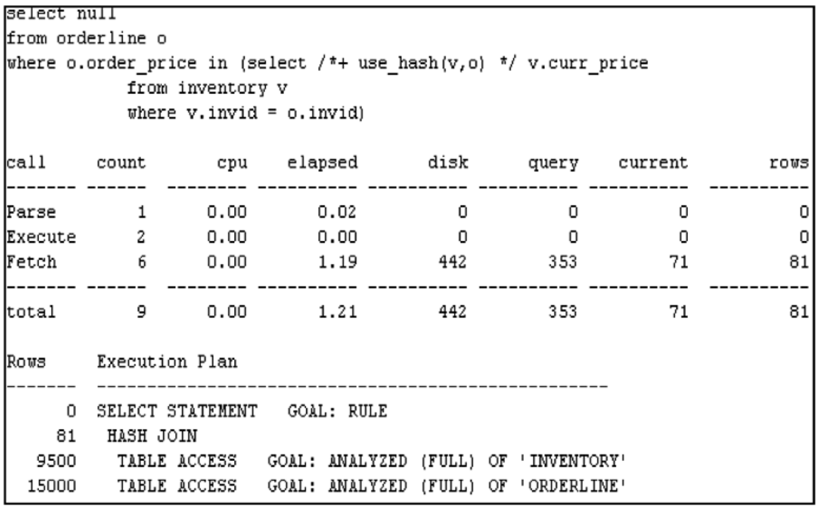
As can be seen in Fig. 14, the execution plan generated for a correlated INDEX attribute on a standard nested query is exactly the same as that for a standard join query, meaning that the optimizer has rewritten the query and treated it as a normal join. Therefore, it does not matter whether or not a Hint is placed in the outer or the inner loop, as either one will cause the optimizer to change the execution plan, only with some differences in the statistics.

Fig. 13 clearly explains the advantage of the hash join algorithm in processing this type of join nested query. As can be seen from the number of rows processed in the execution plan, the nested loop join shows an extensive scan (30,000) in the *Inventory* table while only 81 records are retrieved and joined in the end. On the other



```
select null
from orderline o
where o.order_price in (select v.curr_price
           from inventory v
           where v.invid = o.invid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.08          0          0          0           0
Execute      1      0.00       0.01          0          0          0           0
Fetch        6      0.00       1.53        555      30230          4          81
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        8      0.00       1.62        555      30230          4          81

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
     81   NESTED LOOPS
  15001    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'ORDERLINE'
     81    TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'INVENTORY'
  30000     INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'INVENTORY_INVID_PK'(UNIQUE)
```

(a) No-Hint



```
select null
from orderline o
where o.order_price in (select /*+ use_hash(v,o) */ v.curr_price
           from inventory v
           where v.invid = o.invid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.02          0          0          0           0
Execute      2      0.00       0.00          0          0          0           0
Fetch        6      0.00       1.19        442        353         71          81
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        9      0.00       1.21        442        353         71          81

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
     81   HASH JOIN
   9500    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'INVENTORY'
  15000    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'ORDERLINE'
```

(b) The Best Hint – *use_hash*

Fig. 13. Comparison of No-Hint and Hint optimized trace reports for Correlated INDEX attribute on IN clause nested query structure with inner loop: primary table.

hand, the hash join algorithm scans only the *Inventory* table (9500 records) once into the memory buffer and uses the established hash table in the memory to process the query, thus eliminating an unnecessary scan of the table. Although almost 90% of the execution plans use the same join method, that is the hash join, each of them actually differs in some aspects such as the processing time and the number of disk reads. It is concluded that adding the *use_hash* Hint in the inner loop is the optimal solution. This is because the generated execution plan is the fastest, which is 1.21 s as shown in Fig. 13b. Moreover, the number of disk and memory reads have been reduced to only 442 and 424 respectively, compared with 555 and 30,234 reads initially.

As shown in Fig. 14, there are two Hints used in a correlated NON-INDEX attribute in the *IN* clause nested query with the inner primary table compared with the No-Hint approach. The results have shown that either *use_hash* Hint or *index_join* Hint has reduced the disk read and elapsed time up to 20% from No-Hint. However, as depicted in Fig. 15, the result has shown that the No-Hint approach is better for a correlated NON-INDEX attribute on *IN* clause nested query with inner foreign table compared with all Hints experimented.

For the non-correlated INDEX attribute on a standard nested query using unique inner SELECT, the adding of the *use_hash* Hint on the inner loop has invoked a more efficient join algorithm (hash join) that will result in better query processing. As shown in Fig. 16, applying *use_hash* Hint has decreased the memory read up to 93% when compared with the No-Hint approach. Moreover, the elapsed time using *use_hash* Hint also decreased up to 50% in comparison with the No-Hint approach.

As shown in Fig. 17, we tested all Hints experimented on non-correlated INDEX attribute on a standard nested query against the No-Hint approach. In general, all Hints experimented have faster elapsed time processes than No-Hint. Moreover, *use_merge* Hint is surprisingly faster than that of the *use_hash* Hint. This has made *use_merge* Hint is the best Hint of this type of nested query.

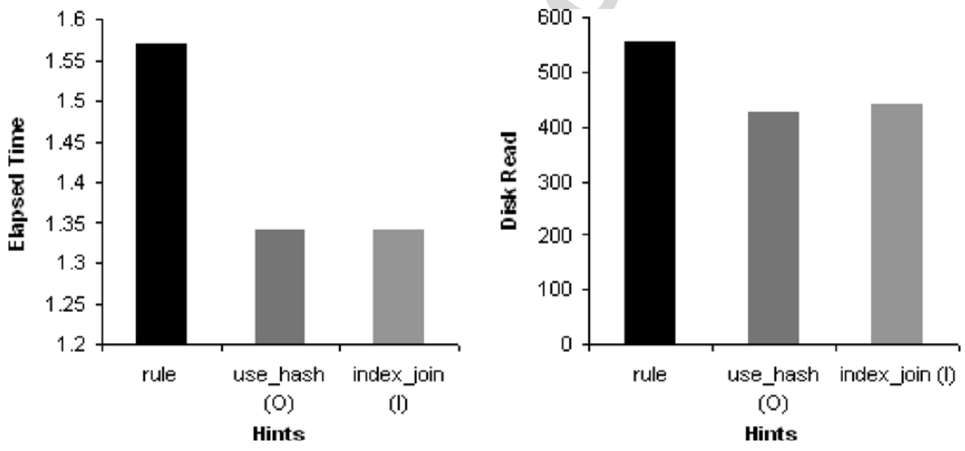


Fig. 14. Comparison of the Hints on the Correlated NON-INDEX attribute on IN clause nested query with inner primary table.

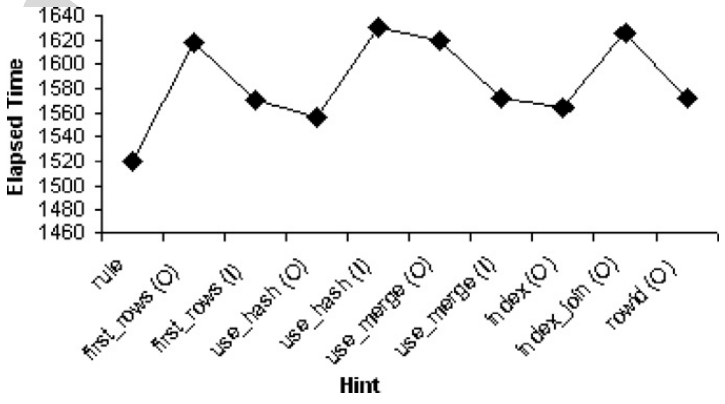


Fig. 15. Comparison of all Hints experimented on the Correlated NON-INDEX attribute on IN clause nested query with inner foreign table.

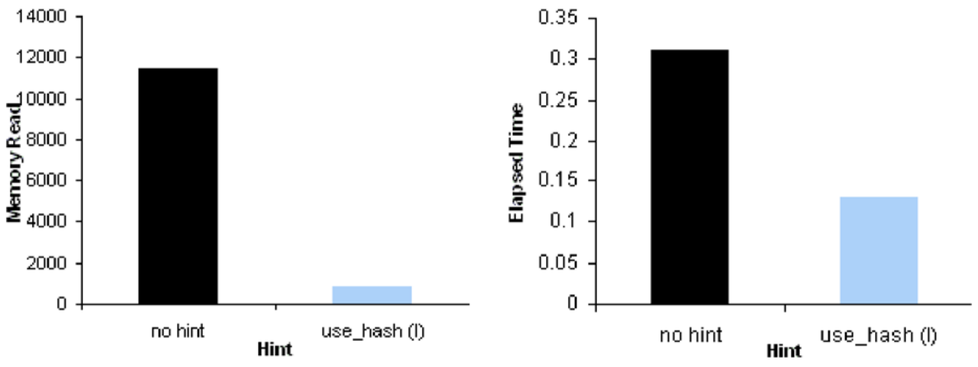


Fig. 16. Comparison of the Hints on the Non-correlated INDEX attribute on IN clause nested queries (Unique Inner SELECT).

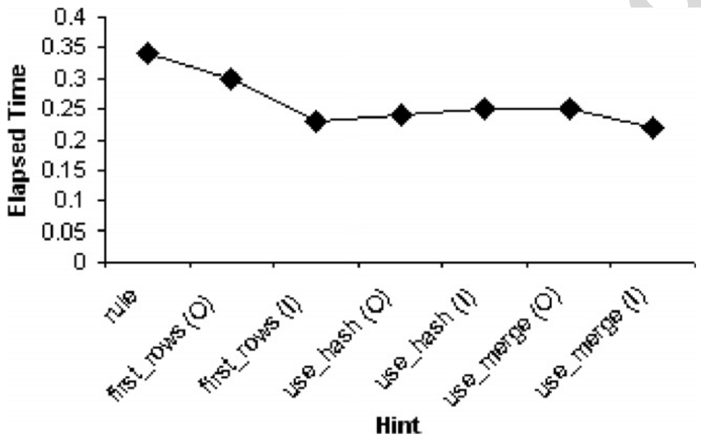


Fig. 17. Comparison of all Hints on Non-correlated NON-INDEX attribute on IN clause nested query.

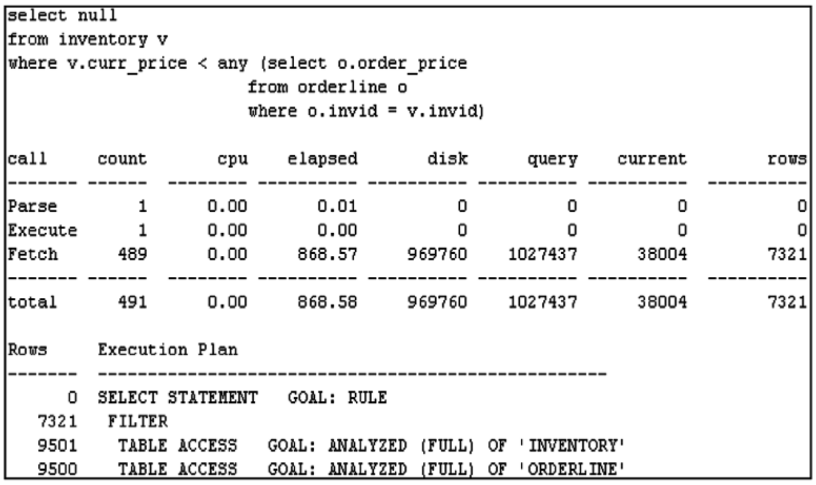
5.2.2. *ANY/ALL clause nested query results*

The Correlated INDEX attribute on a Non-Equality Nested query (ANY clause) structure would normally invoke a full-table scan, so, Oracle has replaced the nested select with the FILTER operation in sync with full-table scan on both *Inventory* and *Orderline* tables. Unfortunately, the all-tested Hints are not generating a more efficient execution plan to process this poorly structured nested query; they managed only to contribute a minor improvement to the processing speed. As shown in Fig. 18, it is found that the *use_merge* Hint is relatively the best of all, with a processing time of 815 s compared with the initial 868.58 s. Besides, the number of disk reads has also been reduced from 969,760 to 967,261 reads.

As shown in Fig. 19, although the No-Hint approach has eliminated the VIEW operation (nested select), it did not utilize the index that exists in the *Staff* table. As a result, the two full-table scans have caused a dramatic downgrade to system performance with a number of disk reads of 23,704. On the other hand, the Hints optimization detects the presence of the index immediately and then replaces the resource intensive full-table scan with an index full scan without any hesitation. After analysing the results from different Hints, the *index_ffs* Hint appears to be superior to the other Hints (see Fig. 20).

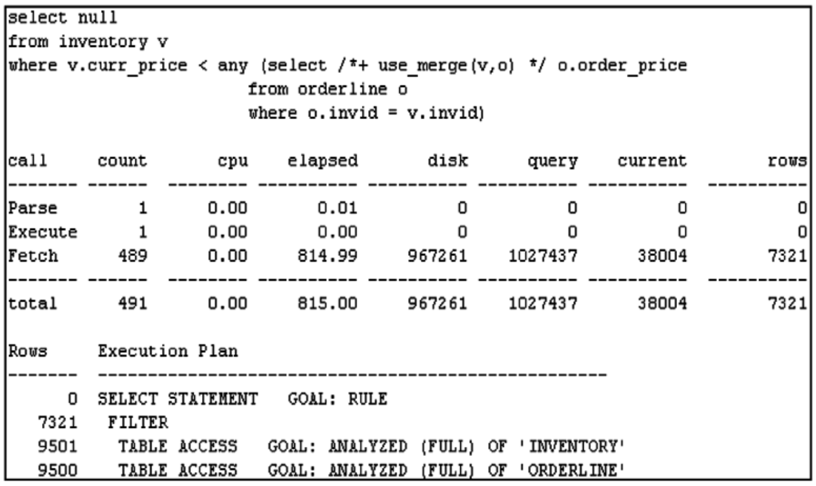
As can be seen in Fig. 20, most of the Hints added have an elapsed time of around 16 s, while the *index_ffs* Hint has an outstanding performance of 14.24 s. This difference is probably caused by a slight change in the access path to the *Staff* table. As shown in Fig. 20b, the optimal Hint invokes an index *fast* full scan operation, which is different from the other Hints that use the index full scan. As a result, the utilisation of index has upturned the number of disk reads from 23,704 to only 185. Moreover, the expected situation, where the use of index should be followed by an increase in memory usage, did not happen. Instead, it has declined by about 33%.

When a nested query join does not involve an indexed attribute, the No-Hint and Hints optimization techniques tend to generate an identical execution plan as shown in Fig. 21. The absence of an index also leaves the



```
select null
from inventory v
where v.curr_price < any (select o.order_price
                          from orderline o
                          where o.invid = v.invid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      489      0.00     868.57     969760    1027437      38004        7321
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      491      0.00     868.58     969760    1027437      38004        7321

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
   7321   FILTER
   9501    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'INVENTORY'
   9500    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'ORDERLINE'
```

(a) No-Hint



```
select null
from inventory v
where v.curr_price < any (select /*+ use_merge(v,o) */ o.order_price
                          from orderline o
                          where o.invid = v.invid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      489      0.00     814.99     967261    1027437      38004        7321
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      491      0.00     815.00     967261    1027437      38004        7321

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
   7321   FILTER
   9501    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'INVENTORY'
   9500    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'ORDERLINE'
```

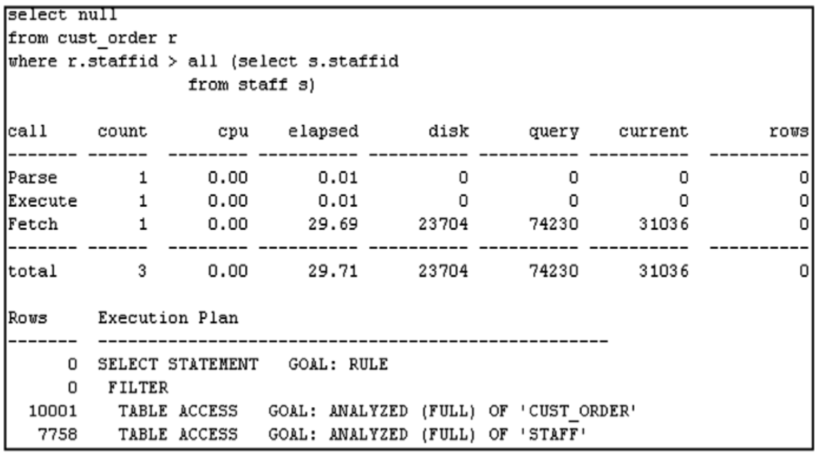
(b) The Best Hint – *use_merge*

Fig. 18. Comparison of No-Hint and Hint optimized trace reports for Correlated INDEX attribute on ANY clause nested query.

optimizer with no choice but to perform a full-table scan on both tables. Therefore, the Hints approach cannot contribute any significant improvement to this type of processing. It is found that *first_rows* Hint causes the optimizer to produce relatively the fastest execution plan (0.49) compared with the other Hints, which is around 0.5 s and there is also a slight improvement in the number of disk reads. Hence, it can be concluded that the Hints approach is still better than the No-Hint in query processing although it did not make significant difference in this case.

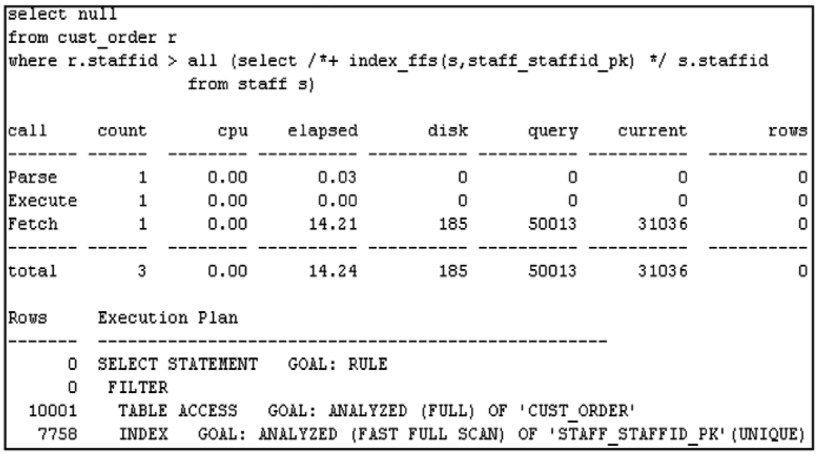
5.2.3. *EXISTS clause nested query results*

There are a number of Hints that perform equally well in optimizing this nested query, which are: (i) *use_hash*, (ii) *use_merge,* and (iii) *rowid*. Interestingly, they all have produced exactly the same execution plans as the original one, which was generated by the No-Hint approach. The graph in Fig. 22 compares the performances of these Hints. As can be seen, the *use_merge* Hint produces the fastest execution plan with 0.10 s, compared with that of the *use_hash* and the *rowid* Hints, which are only 0.01 s slower. On the other hand, *rowid* demonstrates the most efficient resource usage in terms of the physical disk access. Therefore, it seems that there is more than one optimal Hint for this type of nested query and decisions should be made according to specific aims. For example, if the response time is the main issue, the *use_merge* Hint should be chosen, while the *rowid* Hint would be more suitable if the disk access is considered as a more expensive cost component.



```
select null
from cust_order r
where r.staffid > all (select s.staffid
                 from staff s)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.01          0          0          0           0
Fetch        1      0.00      29.69      23704      74230      31036           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00      29.71      23704      74230      31036           0

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
      0   FILTER
  10001    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST_ORDER'
   7758    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF'
```

(a) No-Hint



```
select null
from cust_order r
where r.staffid > all (select /*+ index_ffs(s,staff_staffid_pk) */ s.staffid
                 from staff s)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.03          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00      14.21        185      50013      31036           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00      14.24        185      50013      31036           0

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
      0   FILTER
  10001    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST_ORDER'
   7758    INDEX   GOAL: ANALYZED (FAST FULL SCAN) OF 'STAFF_STAFFID_PK'(UNIQUE)
```

(b) The Best Hint – *index_ffs*

Fig. 19. Comparison of No-Hint and Hint optimized trace reports for Non-correlated INDEX attribute on ALL clause nested query: unique inner SELECT.

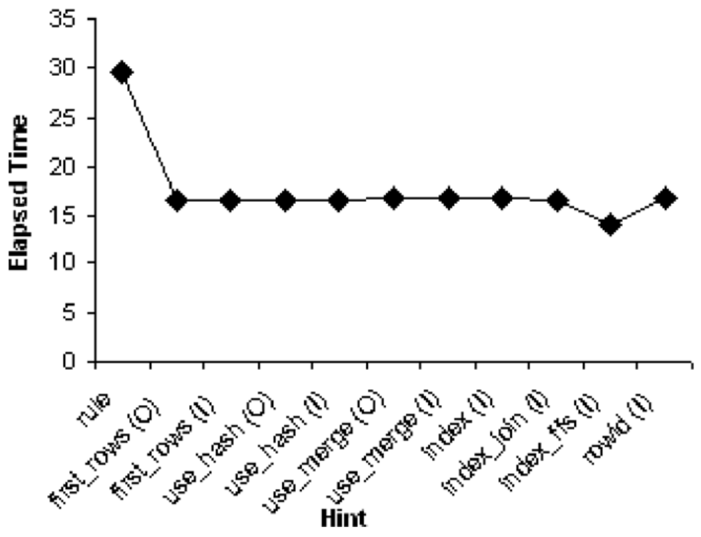
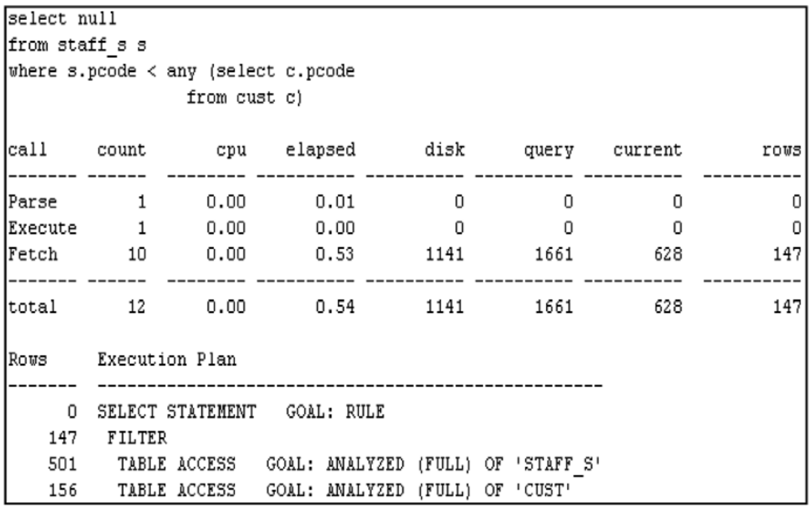


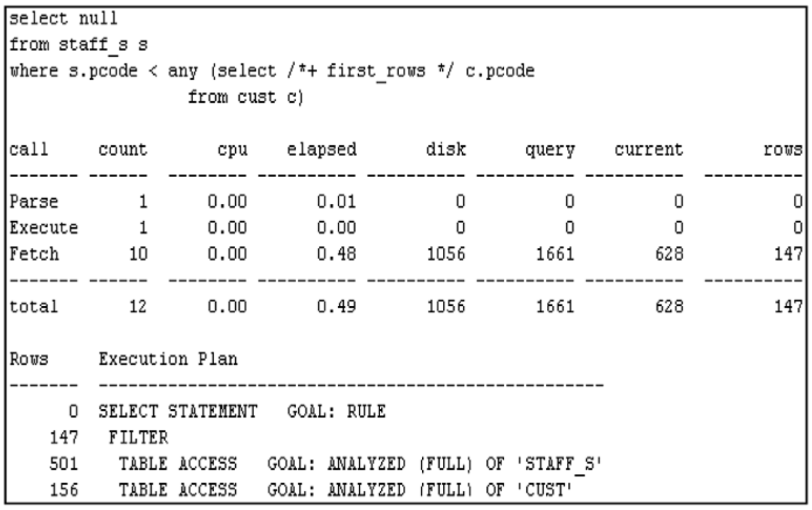
Fig. 20. Comparison of all Hints experimented on Non-correlated INDEX attribute on ALL clause nested query: unique inner SELECT.

As mentioned earlier, Oracle may not be able to drive a given query with a proper table although it can automatically transform certain nested queries into standard joins. This scenario can be seen in Fig. 23a where the *Cust_order* table should really be the outer table since its join attribute is non-indexed. Therefore, it is



```
select null
from staff_s s
where s.pcode < any (select c.pcode
                     from cust c)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       10      0.00       0.53       1141       1661        628         147
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       12      0.00       0.54       1141       1661        628         147

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
    147   FILTER
    501    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF_S'
    156    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST'
```

(a) No-Hint



```
select null
from staff_s s
where s.pcode < any (select /*+ first_rows */ c.pcode
                     from cust c)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       10      0.00       0.48       1056       1661        628         147
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       12      0.00       0.49       1056       1661        628         147

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
    147   FILTER
    501    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF_S'
    156    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST'
```

(b) The Best Hint – *first_rows*

Fig. 21. Comparison of No-Hint and Hint optimized trace report on Non-Correlated NON-INDEX attribute on ANY clause nested query.

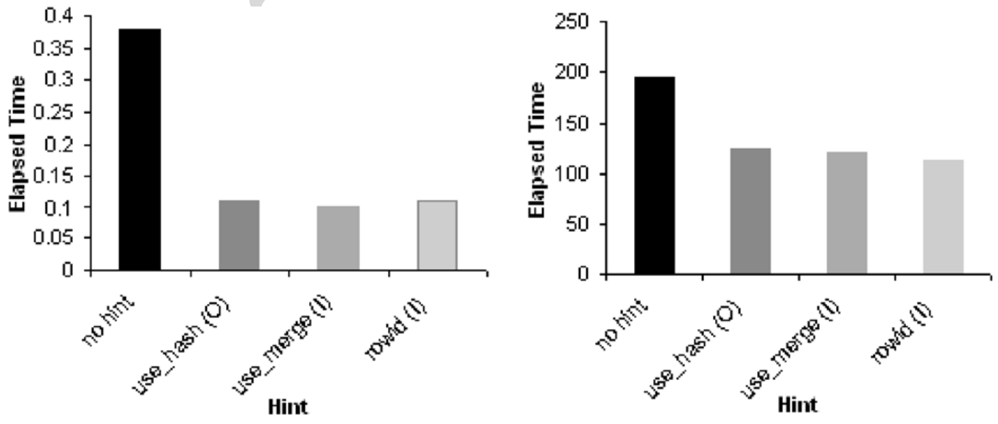
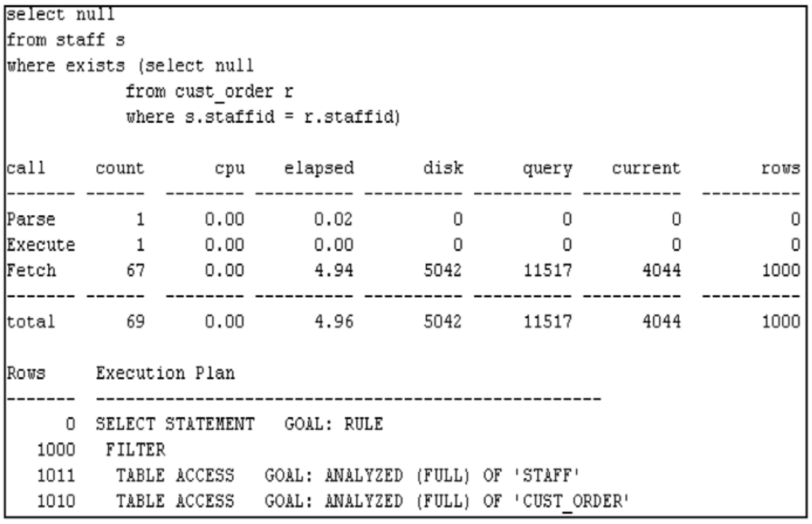


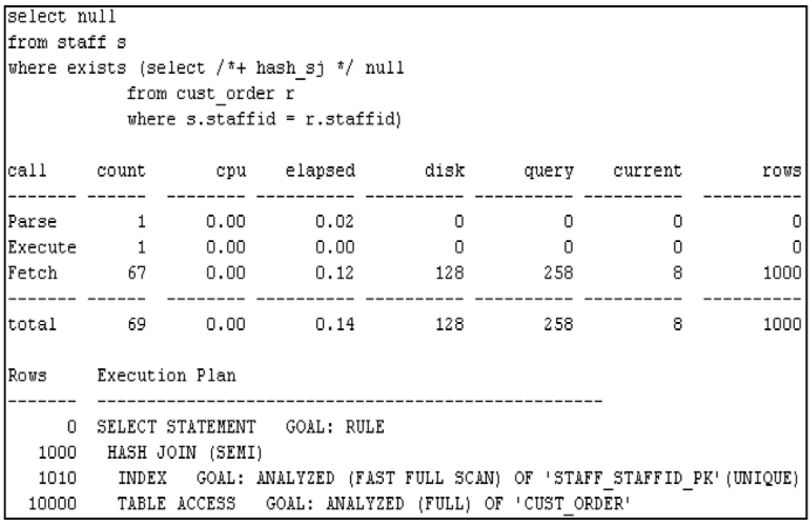
Fig. 22. Comparison of the Hints experimented in Correlated INDEX attribute on EXISTS clause nested query: inner primary table.

important to optimise a nested query with every available means while using Hint as only one of the solutions. It is found that the *hash_sj* Hint has the most outstanding performance by being able to reduce the processing time to only 0.14 s compared to the initial 4.96 s. This significant reduction is clearly shown in Fig. 24 when



```
select null
from staff s
where exists (select null
            from cust_order r
            where s.staffid = r.staffid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       67      0.00       4.94       5042      11517       4044        1000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       69      0.00       4.96       5042      11517       4044        1000

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
   1000   FILTER
   1011    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF'
   1010    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST_ORDER'
```

(a) No-Hint



```
select null
from staff s
where exists (select /*+ hash_sj */ null
            from cust_order r
            where s.staffid = r.staffid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       67      0.00       0.12        128        258          8        1000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       69      0.00       0.14        128        258          8        1000

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
   1000   HASH JOIN (SEMI)
   1010    INDEX   GOAL: ANALYZED (FAST FULL SCAN) OF 'STAFF_STAFFID_PK'(UNIQUE)
  10000    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST_ORDER'
```

(b) The Best Hint – *hash_sj*

Fig. 23. Comparison of rule-based and Hint optimized trace report of Correlated INDEX attribute on EXISTS Clause Nested Query (inner Foreign Table).

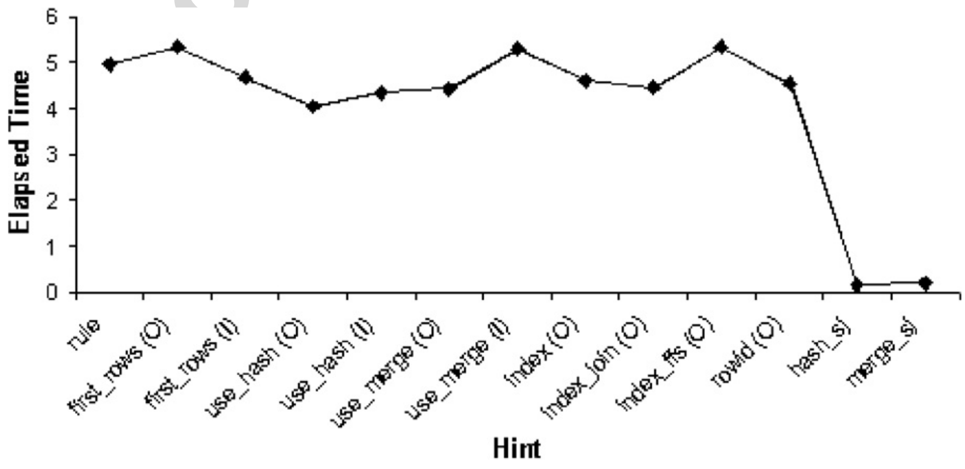


Fig. 24. Comparison of all Hints experimented on Correlated INDEX attribute on EXISTS nested query (inner foreign table).

compared with the other Hints that perform equally poorly as the original execution plan. Note that the *merge_sj* Hint can also produce a very fast execution plan being only 0.05 s slower than the optimal Hint.

As shown in Fig. 23b, the best Hint has invoked a semi hash join, which is designed specifically for nested queries using the EXISTS clause, to replace the FILTER operation. Although the *Staff* table remains as the outer table, the optimizer has utilized the index by using an index fast full scan instead of the time-consuming full-table scan. Apparently, this whole new execution plan is more suitable for this type of correlated nested queries due to its ability to significantly improve the overall performance. The improved cost components include the processing time, and the number of physical disk and memory reads as can be seen from Fig. 23. In Fig. 24, all Hints experimented have shown that only *hash_sj* and *merge_sj* hints have the best elapsed time, being less than 1 s.

As can be seen from the elapsed columns in Fig. 25b and c, both the *use_merge* and the *merge_sj* Hints need the same period of time (0.03 s) to process the given non-key correlated nested query that uses the EXISTS clause. However, the same time frame is achieved with different execution plans. Under the *use_merge* Hint, Fig. 25b shows that the execution plan generated is exactly the same as that from No-Hint. Interestingly, its processing time and disk access are 86% and 37% better respectively.

On the other hand, the *merge_sj* Hint actually causes the optimizer to invoke a different operation (merge semi-join) to process the query. While it results in the same improvement in the processing speed as the first Hint did, the number of disk reads is nearly the same as the original plan with only one disk read difference. This is likely to be caused by the sorting operation on two tables, which requires a large sort area [19]. Nonetheless, this execution plan is better in memory consumption by reading only 33 data blocks from the memory buffer compared to the initial 223 reads. On balance, the *use_merge* Hint may be a better choice due to its ability to cut down the more expensive component, which is the physical disk read.

5.2.4. NOT EXISTS clause nested query results

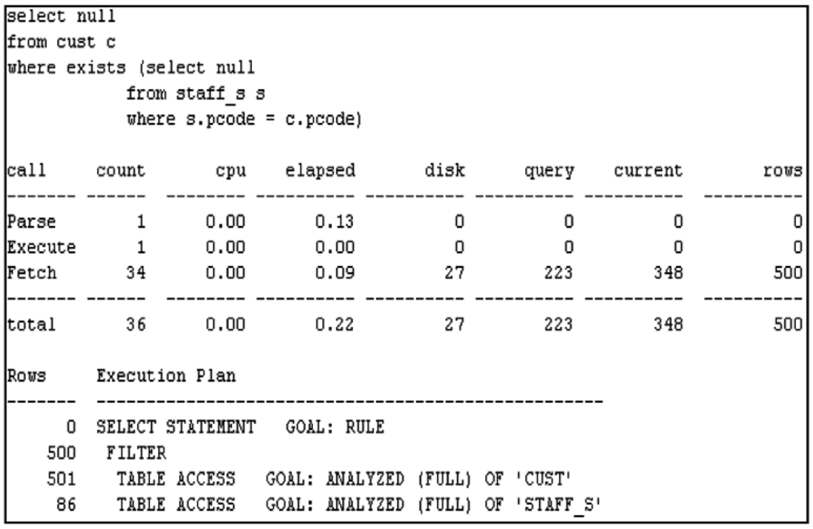
The No-Hint approach appears to be slightly better than the Hints optimization technique on the Correlated INDEX attribute on the NOT EXISTS Clause Nested Query with the inner primary table. While all the execution plans generated by the Hints are exactly the same as in Fig. 26, they need more than 0.46 s to process the given nested query. There is one Hint (*index_ffs*) that managed to replace the index unique scan on the *Staff* table with an index fast full scan. However, the processing time soared to 13 s. In the meantime, both the *hash_aj* and the *merge_aj* Hints do not work for this join structure, meaning that the anti-join operation cannot be invoked.

As usual, the No-Hint approach uses the full-table scan to process the transformed correlated nested query. On the other hand, every Hint added to the SQL statement instructs the system to utilize the available index, which then improves the processing time. After comparing the performance of all Hints, it is found that the execution plan produced by the *use_hash* Hint is the fastest. As can be seen in Fig. 27b, while no hash join is invoked, the index in *Staffid* attribute has been utilized to access the *Staff* table instead of a full-table scan, which is slower. However, this response time improvement is at the expense of disk reads. Also, the disk column shows an extra 332 disk reads compared with that in Fig. 27a. Therefore, it cannot be concluded that adding the *use_hash* Hint to the correlated INDEX attribute on a NOT EXISTS clause nested query on inner foreign table is definitely the best solution - unless response time is the main concern.

As shown in Fig. 28, the adding of Hints did not produce a significant improvement to the correlated NON-INDEX attribute on the NOT EXISTS clause nested query. Moreover, neither the *hash_aj* nor the m *erge_aj* Hints should work for nested queries that use the NOT EXISTS clause, meaning that no anti-join operation would be invoked in the execution plan, the latter Hint appears to have the fastest processing time among the other Hints. This means that improvement has been made to the processing speed without changing the execution plan. The other improvement shown in Fig. 28b is a minor reduction in the number of disk reads.

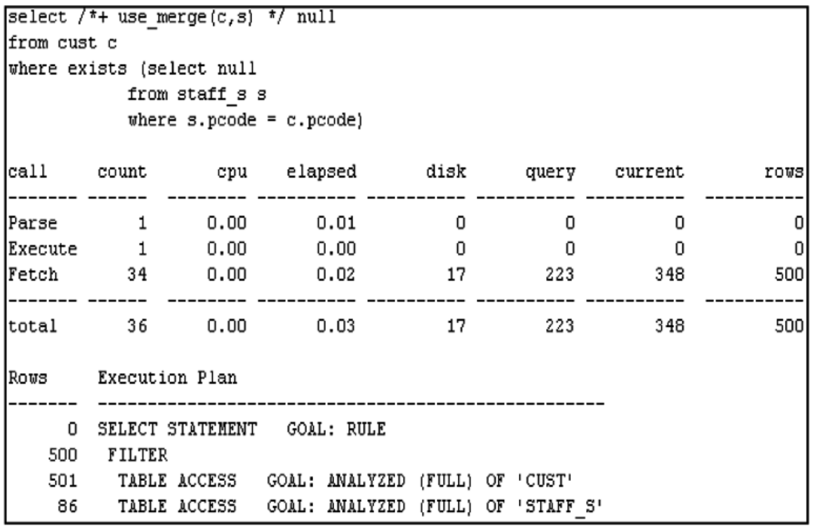
5.2.5. NOT IN clause nested query results

Experiment results show that no single Hint was able to change the execution plan for the correlated INDEX attribute in the NOT IN Clause Nested Query on the inner primary table. It is insufficient to bring significant improvement to the processing performance by adding Hints solely to this type of nested query



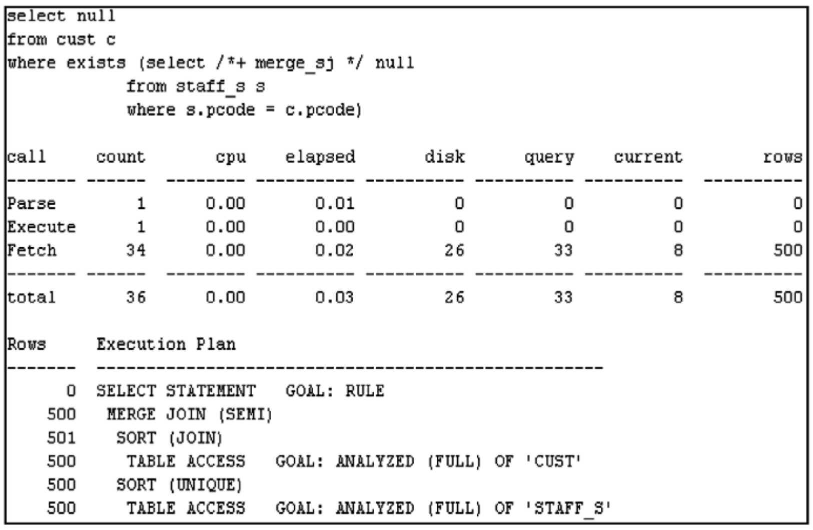
```
select null
from cust c
where exists (select null
           from staff_s s
           where s.pcode = c.pcode)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.13          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       34      0.00       0.09         27        223        348         500
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       36      0.00       0.22         27        223        348         500

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
    500   FILTER
    501    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST'
     86    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF_S'
```

(a) No-Hint



```
select /*+ use_merge(c,s) */ null
from cust c
where exists (select null
           from staff_s s
           where s.pcode = c.pcode)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       34      0.00       0.02         17        223        348         500
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       36      0.00       0.03         17        223        348         500

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
    500   FILTER
    501    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST'
     86    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF_S'
```

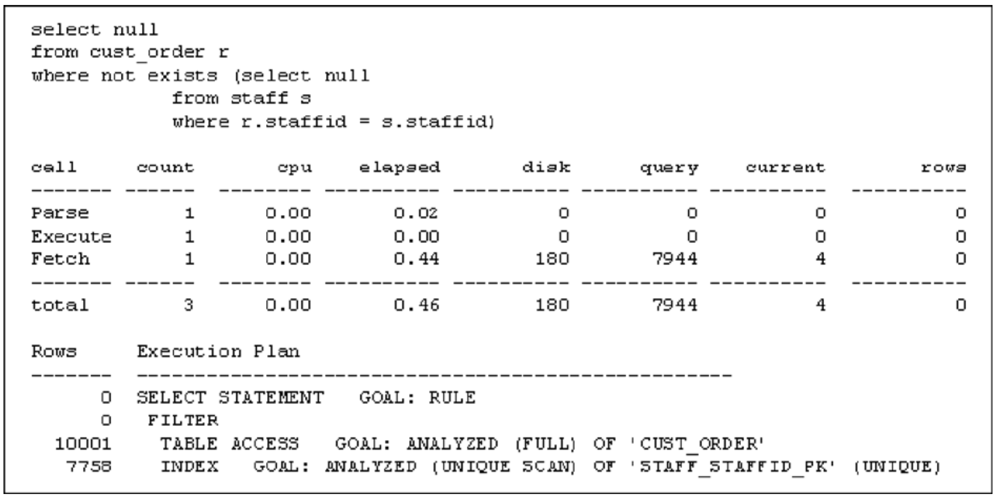
(b) The *use_merge* Hint



```
select null
from cust c
where exists (select /*+ merge_sj */ null
           from staff_s s
           where s.pcode = c.pcode)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       34      0.00       0.02         26         33          8         500
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       36      0.00       0.03         26         33          8         500

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
    500   MERGE JOIN (SEMI)
    501    SORT (JOIN)
    500     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST'
    500    SORT (UNIQUE)
    500     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF_S'
```

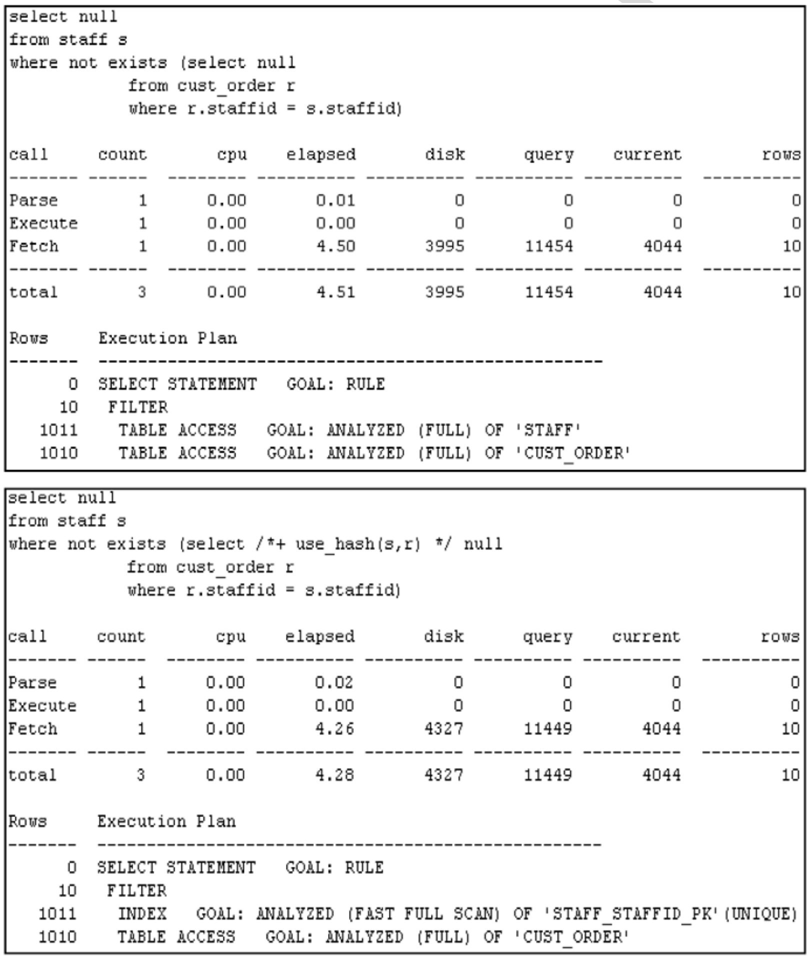
(c) The *merge_sj* Hint

Fig. 25. Comparison of No-Hint and Hint optimized trace reports for Correlated NON-INDEX attribute on EXISTS nested query.



```
select null
from cust_order r
where not exists (select null
            from staff s
            where r.staffid = s.staffid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.44        180       7944          4           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.46        180       7944          4           0

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
      0   FILTER
  10001    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST_ORDER'
   7758    INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'STAFF_STAFFID_PK' (UNIQUE)
```

Fig. 26. The best TKPROF result No-Hint for the Correlated INDEX attribute on NOT EXISTS clause nested query: inner primary table.



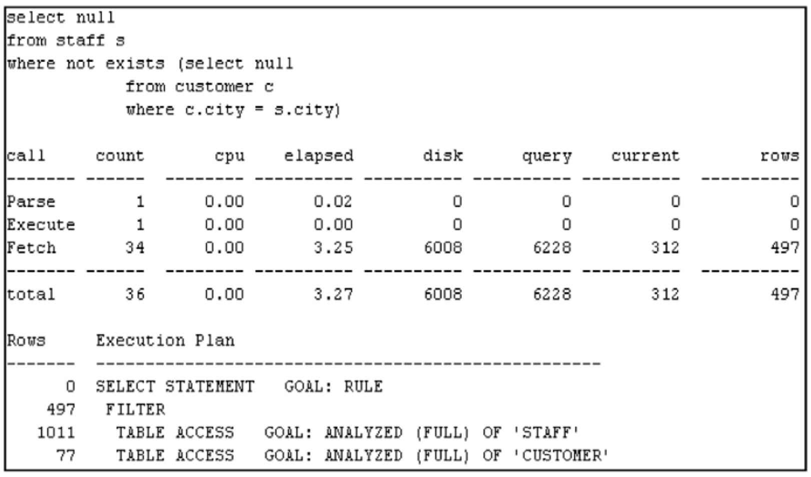
```
select null
from staff s
where not exists (select null
            from cust_order r
            where r.staffid = s.staffid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       4.50       3995      11454       4044          10
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       4.51       3995      11454       4044          10

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
     10   FILTER
   1011    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF'
   1010    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST_ORDER'
```

```
select null
from staff s
where not exists (select /*+ use_hash(s,r) */ null
            from cust_order r
            where r.staffid = s.staffid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       4.26       4327      11449       4044          10
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       4.28       4327      11449       4044          10

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
     10   FILTER
   1011    INDEX   GOAL: ANALYZED (FAST FULL SCAN) OF 'STAFF_STAFFID_PK'(UNIQUE)
   1010    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST_ORDER'
```

Fig. 27. Comparison of No-Hint and Hint optimized trace reports for Correlated INDEX attribute on NOT EXISTS clause nested query: inner foreign table.

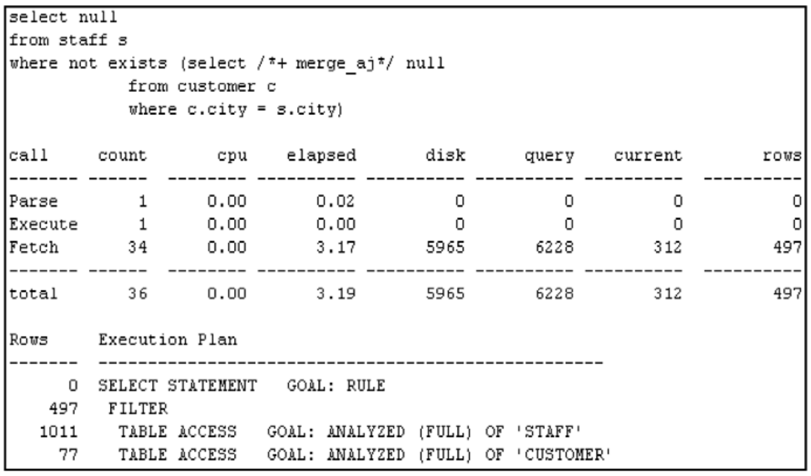
structure for the purpose of optimisation. As can be seen in Fig. 29b, the number of the expensive disk reads is almost half of the original, which is then followed by a 40% reduction in processing time.

Although the changes to the performance are not as significant, the *use_hash* Hint is still relatively the best compared with the other Hints and the No-Hint optimisation. Therefore, it is worthwhile to add Hints to this



```
select null
from staff s
where not exists (select null
            from customer c
            where c.city = s.city)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       34      0.00       3.25       6008       6228        312         497
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       36      0.00       3.27       6008       6228        312         497

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
    497   FILTER
   1011    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF'
     77    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUSTOMER'
```

(a) No-Hint



```
select null
from staff s
where not exists (select /*+ merge_aj*/ null
            from customer c
            where c.city = s.city)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       34      0.00       3.17       5965       6228        312         497
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       36      0.00       3.19       5965       6228        312         497

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
    497   FILTER
   1011    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF'
     77    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUSTOMER'
```

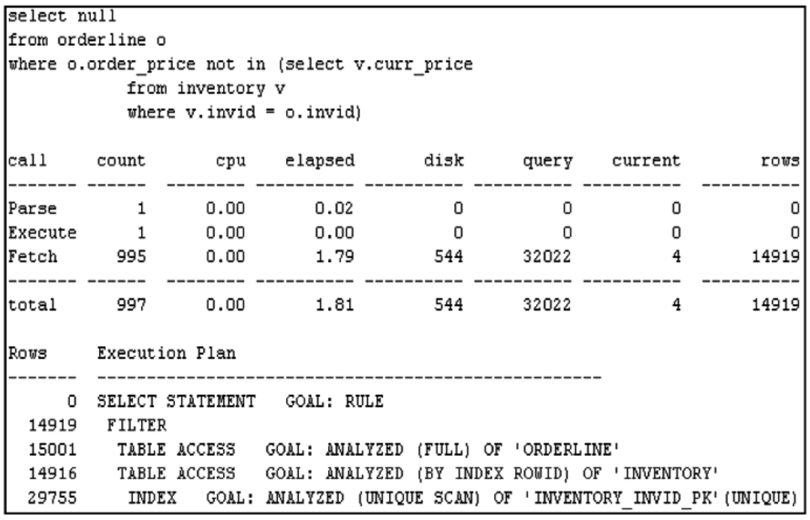
(b) The Best Hint – *merge_aj*

Fig. 28. Comparison of No-Hint and Hint optimized trace reports for the Correlated NON-INDEX attribute on NOT EXISTS clause nested query.

type of key-correlated nested queries using the IN clause, if users wish to avoid the inconvenience of rewriting the query, or if they are unfamiliar with other optimisation techniques.

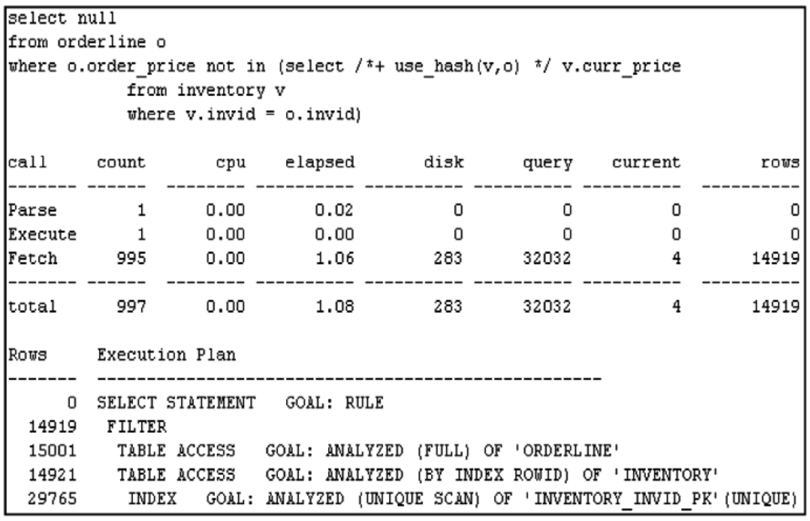
Once again, instructions from Hints alone are inadequate as a means for the system to generate any better execution plans than the No-Hint approach. On the correlated INDEX attribute in the NOT IN Clause Nested Query with the inner foreign table, the best Hint shown in Fig. 30b is in terms of response time. As can be seen, the plan produced with the help of the Hint is 123 s faster than the original one. However, this improvement is at the great expense of disk reads. The disk column shows an increase from 1,743,303 reads to 2,000,669 reads, which is an extra of 257,366 reads. This means, the system would need to perform additional physical disk reads of 2092 in order to improve response time by only 1 s. On balance, it seems that the costs involved have outweighed the benefits received.

The three Hints shown in Fig. 31 on the Correlated INDEX attribute on the NOT IN Clause Nested Query with the inner primary table have the best performance of all with a reduction of almost 100% in both the processing time and the number of disk reads compared with the original one generated by the No-Hint approach. It is observed that the *use_hash* and the *merge_aj* Hint have the same and the fastest processing speed (1.35 s), while the *hash_aj* Hint uses the least resource (disk reads) and is only 0.01 s slower than the other two. Therefore, it can be said that the three Hints perform equally well in optimizing this type of nested query and adding either of them to an SQL statement would produce similar improvements to the system performance. However, it might be more reasonable to employ the *use_hash* Hint, since the other two did not



```
select null
from orderline o
where o.order_price not in (select v.curr_price
            from inventory v
            where v.invid = o.invid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      995      0.00       1.79        544      32022          4       14919
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      997      0.00       1.81        544      32022          4       14919

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
  14919   FILTER
  15001    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'ORDERLINE'
  14916    TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'INVENTORY'
  29755     INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'INVENTORY_INVID_PK'(UNIQUE)
```

(a) No-Hint



```
select null
from orderline o
where o.order_price not in (select /*+ use_hash(v,o) */ v.curr_price
            from inventory v
            where v.invid = o.invid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      995      0.00       1.06        283      32032          4       14919
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      997      0.00       1.08        283      32032          4       14919

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
  14919   FILTER
  15001    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'ORDERLINE'
  14921    TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'INVENTORY'
  29765     INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'INVENTORY_INVID_PK'(UNIQUE)
```

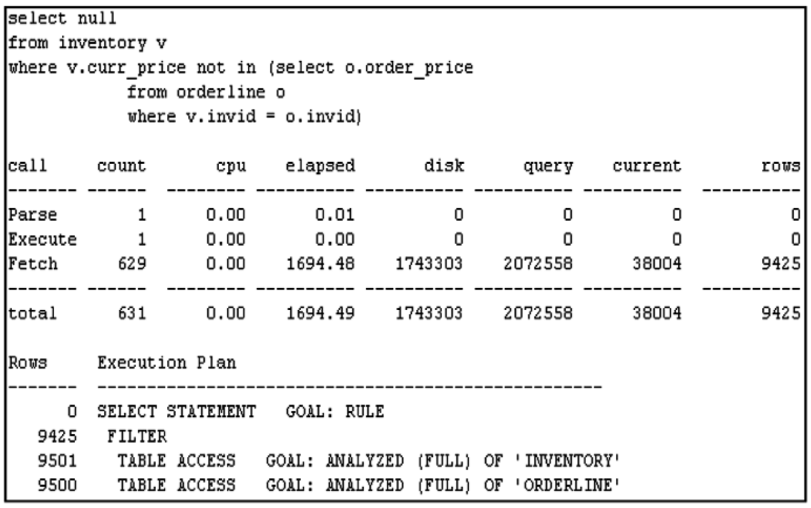
(b) The Best Hint – *use_hash*

Fig. 29. Comparison of No-Hint and Hint optimized trace reports for the Correlated INDEX attribute on NOT IN clause nested query: inner primary table.

actually invoke an anti-join operation, indicating a possibility that they are not designed to be used in this join structure.

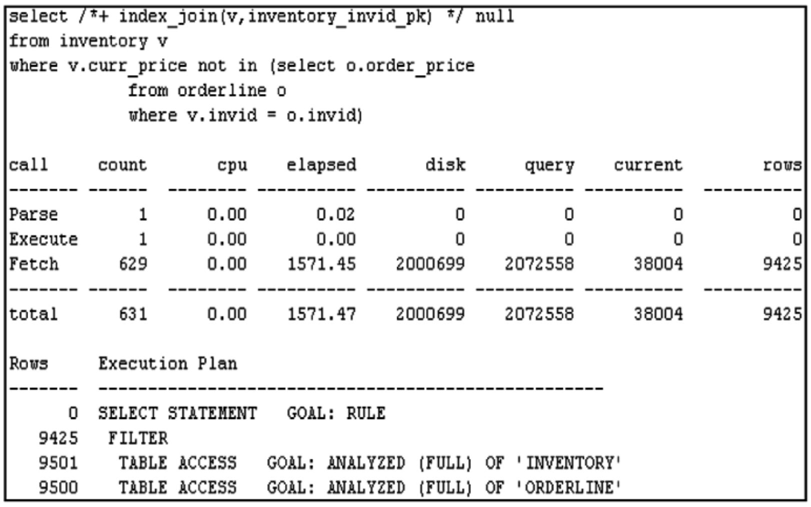
Experiment results of the Correlated INDEX attribute on the NOT IN Clause Nested Query with an inner foreign table show (Fig. 32) that the Hints optimization failed to generate a more efficient execution plan to process this poorly structured nested query. Therefore, both the No-Hint and the Hints optimization have generated identical strategies. Nonetheless, the *merge_aj* Hint is the best in comparison with the others, which managed to contribute some minor improvements to the statistics. As can be seen from Fig. 32, the processing time is now 1587.85 s compared with the initial 1693.05 s, while the number of disk reads has also been reduced from 2,005,487 to 1,999,338 reads.

After comparing the performance of each Hint experimented on a Non-correlated INDEX attribute on an Anti-Nested query (NOT In clause) with the unique inner SELECT, the *index_ffs* Hint as shown in Fig. 33 appears to be able to produce the fastest execution plan with a processing time of 15.62 s, which is almost double that of the original speed. This is because it uses the index fast full scan to access the *Staff* table instead of an index full scan as did the other Hints. The other significant improvements following the change are a 99%



```
select null
from inventory v
where v.curr_price not in (select o.order_price
            from orderline o
            where v.invid = o.invid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      629      0.00    1694.48    1743303    2072558      38004        9425
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      631      0.00    1694.49    1743303    2072558      38004        9425

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
   9425   FILTER
   9501    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'INVENTORY'
   9500    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'ORDERLINE'
```

(a) No-Hint



```
select /*+ index_join(v,inventory_invid_pk) */ null
from inventory v
where v.curr_price not in (select o.order_price
            from orderline o
            where v.invid = o.invid)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      629      0.00    1571.45    2000699    2072558      38004        9425
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      631      0.00    1571.47    2000699    2072558      38004        9425

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
   9425   FILTER
   9501    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'INVENTORY'
   9500    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'ORDERLINE'
```

(b) The Best Hint – *index_join*

Fig. 30. Comparison of No-Hint and Hint optimized trace reports for the Correlated INDEX attribute on NOT IN clause nested query: inner foreign table.

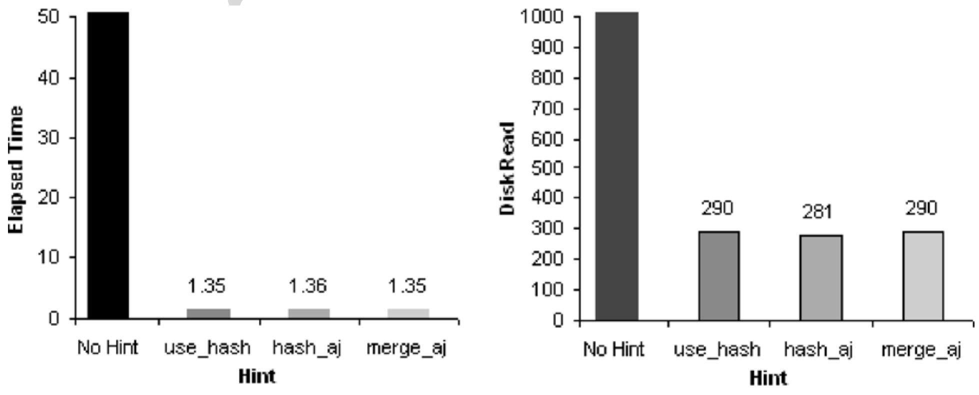


Fig. 31. Comparison of No-Hint and Hints optimized for Correlated INDEX attribute on NOT IN clause nested query: inner primary table.

reduction in the number of disk reads and also a 33% decline in memory reads. Therefore, there is no reason not to add the *index_ffs* Hint to the inner loop of this type of Non-correlated nested queries for optimization purposes.

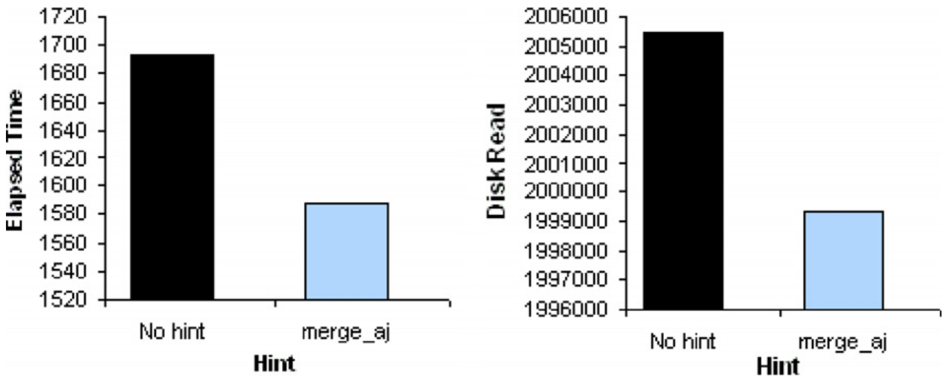
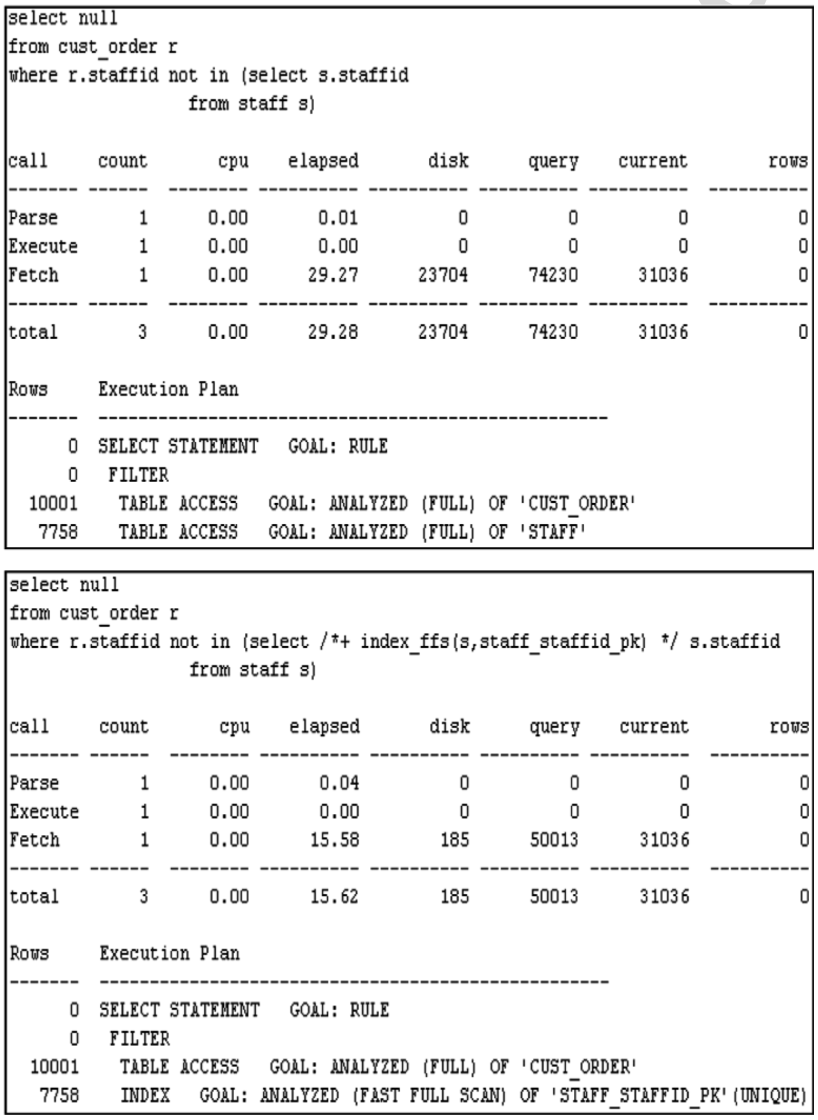


Fig. 32. Comparison of No-Hint and Hints optimized for correlated INDEX attribute on NOT IN clause nested query: inner foreign table.



```
select null
from cust_order r
where r.staffid not in (select s.staffid
                   from staff s)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00      29.27      23704      74230      31036           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00      29.28      23704      74230      31036           0

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
      0   FILTER
  10001    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST_ORDER'
   7758    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF'
```

```
select null
from cust_order r
where r.staffid not in (select /*+ index_ffs(s,staff_staffid_pk) */ s.staffid
                   from staff s)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.04          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00      15.58        185      50013      31036           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00      15.62        185      50013      31036           0

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
      0   FILTER
  10001    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUST_ORDER'
   7758    INDEX   GOAL: ANALYZED (FAST FULL SCAN) OF 'STAFF_STAFFID_PK'(UNIQUE)
```

Fig. 33. Comparison of No-Hint and Hint optimized trace reports for the Non-correlated NOT IN clause nested query: unique inner SELECT.

The No-Hint approach appears to perform better than the Hints optimization technique in the processing of a Non-correlated INDEX attribute on a NOT IN Clause Nested Query with the non-unique inner SELECT. As shown in Fig. 34, although all the elapsed times are below 5 s, the adding of Hint shows further

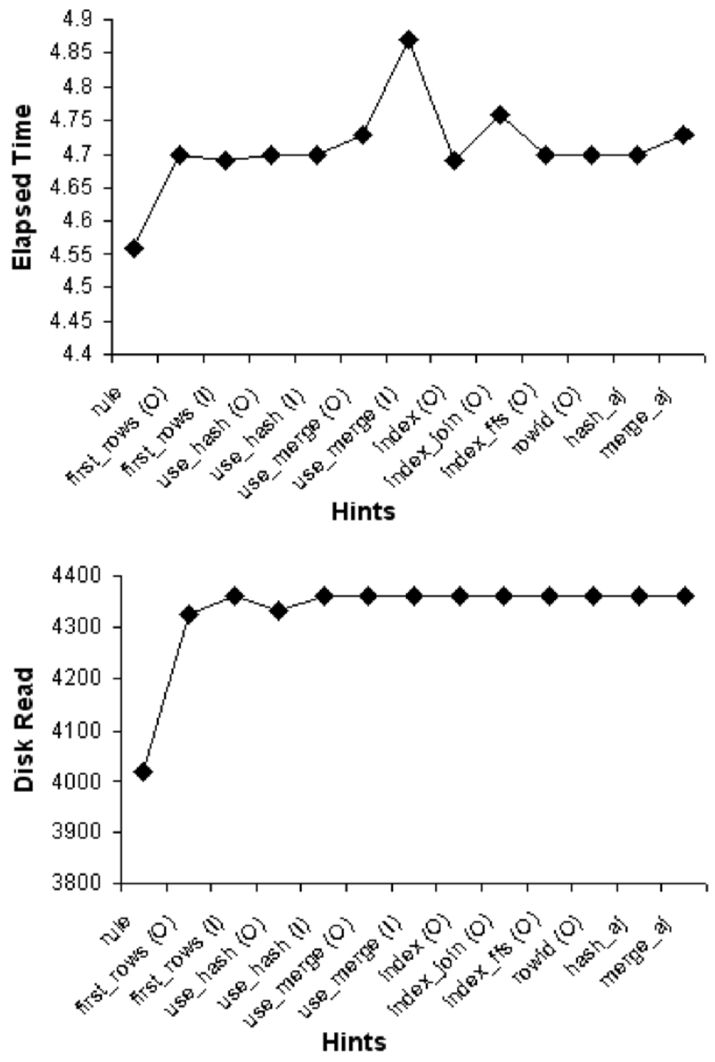


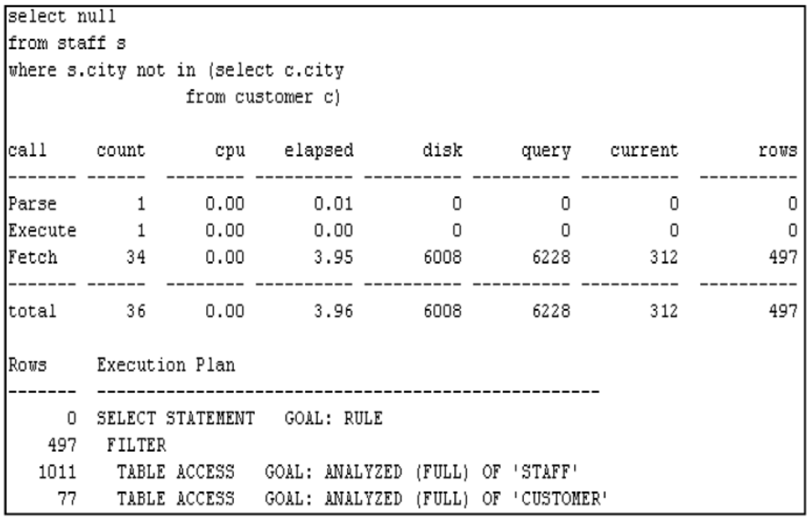
Fig. 34. Comparison of all Hints experimented on the Non-correlated NOT IN clause nested query: non-unique inner SELECT.

deterioration to system performance in terms of both processing time and disk reads. Therefore, it is best to process the Non-correlated INDEX attribute on a NOT IN Clause Nested Query with a non-unique inner SELECT under the rule-based optimization.

When no index is applicable to the Non-correlated NON-INDEX attribute on a NOT IN Clause Nested Query, both the No-Hint and the Hints optimization have generated identical execution plans with a full-table scan of both tables as expected. However, the Hints technique still performs slightly better than the No-Hint approach by contributing minor improvement to the processing time and the number of disk reads. It is found that the *first_rows* Hint added to the outer loop is the best candidate compared with the other Hints by slightly improving the performance as shown in Fig. 35b. This means, the contribution of Hints optimization does not significantly guide the current system to optimize the given nested query.

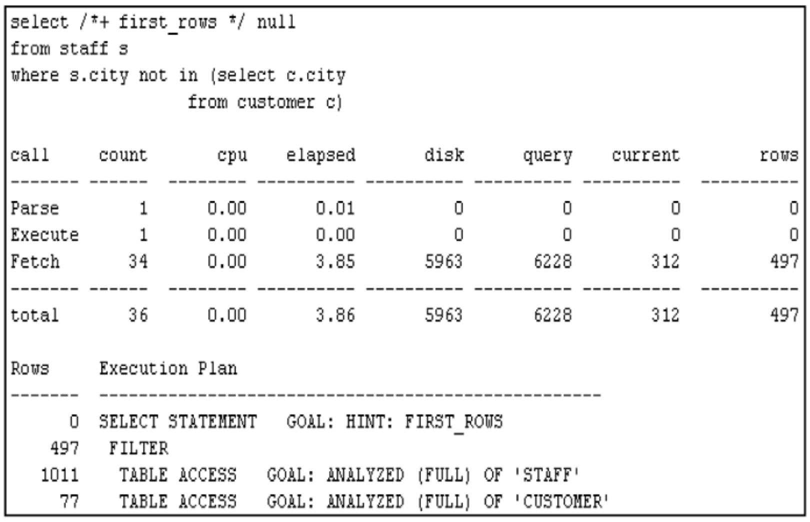
5.3. Discussions

There can be more than one goal when it comes to optimizing a given query. For instance, the main concern of a company running an online business (online transaction processing) will be the response time, as customers or clients would not want to waste their time waiting for web page reloading while a transaction is being processed. Therefore, the DBMS must be able to produce and display the first available result as quickly as possible, although it may be at the expense of resources or overall processing time. On the other hand, tasks such as generating a large report generally require the best throughput (minimum total resource consumption)



```
select null
from staff s
where s.city not in (select c.city
                 from customer c)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       34      0.00       3.95       6008       6228        312         497
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       36      0.00       3.96       6008       6228        312         497

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: RULE
    497   FILTER
   1011    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF'
     77    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUSTOMER'
```

(a) No-Hint



```
select /*+ first_rows */ null
from staff s
where s.city not in (select c.city
                 from customer c)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       34      0.00       3.85       5963       6228        312         497
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       36      0.00       3.86       5963       6228        312         497

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: HINT: FIRST_ROWS
    497   FILTER
   1011    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'STAFF'
     77    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'CUSTOMER'
```

(b) The Best Hint - *first_rows*

Fig. 35. Comparison of No-Hint and Hint optimized trace reports for the Non-correlated NON-INDEX attribute on NOT IN clause nested query.

since responsiveness does not play a key role. That is, while it may need a longer period to display the first record, the overall performance is better.

In this paper, an optimal Hint is chosen based on the fastest overall processing time. However, it is possible that some of the chosen Hints might be resource intensive. Table 3 recommends the type of *Hint* which is best for each individual join structure.

On *IN* clause nested queries results, by applying *hash_sj* Hint on Correlated INDEX attribute with inner loop foreign table structure, *use_merge* Hint on Correlation INDEX attribute with inner loop primary table and Correlation NON-INDEX attribute structure, it has shown those Hints do effectively improve the system performance compared with the No-Hint approach. While on *ANY/ALL Clause nested query* results, the use of *use_hash* Hint either on Correlated INDEX attribute with inner loop primary table, correlated NON-INDEX attribute with inner loop primary table structure or Non-correlated INDEX attribute structure and the use of index Hint on correlation INDEX attribute with inner loop foreign table and the use of *use_merge* Hint on Non-correlated NON-INDEX attribute, has proven that the Hints approach is better than the No-Hint approach. However, on a Correlated NON-INDEX attribute with an inner loop foreign table join structure, none of the optimized Hints approaches is better than the No-Hint approach.

Table 3
Optimal Hint(s) for each join nested query structure

Clause	Join structure	Additional join structure	Optimal Hint (s)
EXISTS	Correlation: INDEX attribute	Inner loop: primary table	*Use_merge* (inner loop)
		Inner loop: foreign table	*Hash_sj* (inner loop)
	Correlation: NON-INDEX attribute	–	*Use_merge* (outer loop)
IN	Correlation: INDEX attribute	Inner loop: primary table	*Use_hash* (inner loop)
		Inner loop: foreign table	*Index* (outer loop)
	Correlation: NON-INDEX attribute	Inner loop: primary table	*Use_hash* (outer loop)
		Inner loop: foreign table	No-Hint
	Non-correlated: INDEX attribute	Inner SELECT: unique primary key	*Use_hash* (inner loop)
		Inner SELECT: non-unique foreign key	*Use_hash* (inner loop)
	Non-correlated: NON-INDEX attribute	–	*Use_merge* (inner loop)
ANY/ALL	Correlation: INDEX attribute	–	*Use_merge* (inner loop)
	Non-correlated: INDEX attribute	Inner SELECT: unique primary key	*Index_ffs* (inner loop)
		Inner SELECT: non-unique foreign key	*Use_hash* (inner loop)
	Non-correlated: NON-INDEX attribute	–	*First_rows* (inner loop)
NOT EXISTS	Correlation: INDEX attribute	Inner loop: primary table	No-Hint
		Inner loop: foreign table	*Use_hash* (inner loop)
	Correlation: NON-INDEX attribute	–	*First_rows* (inner loop)
NOT IN	Correlation: INDEX attribute	Inner loop: primary table	*Use_hash* (inner loop)
		Inner loop: foreign table	No-Hint
	Correlation: NON-INDEX attribute	Inner loop: primary table	*Use_hash* (inner loop)
			Merge_aj (inner loop)
		Inner loop: foreign table	*Merge_aj* (inner loop)
	Non-correlated: INDEX attribute	Inner SELECT: unique primary key	*Index_ffs* (inner loop)
		Inner SELECT: non-unique foreign key	No-Hint
	Non-correlated: NON-INDEX attribute	–	*First_rows* (outer loop)

Moreover, for the EXISTS clause nested queries results, only two Hints are most suitable, these being the *use_merge* Hint for correlation of the INDEX attribute with the inner loop primary table and correlation of the NON-INDEX attribute; and, the *hash_sj* Hint for correlation of the INDEX attribute with the inner loop foreign table. Finally, for the NOT EXISTS clause nested queries results on Correlation INDEX attribute with inner loop foreign table, the *use_hash* Hint is most suitable. However, no optimized Hint is best for the correlation of the INDEX attribute with the inner loop primary table by using the NOT EXISTS clause. The *first_rows* Hint is optimizing correlation NON-INDEX ATTRIBUTE structure on the NOT EXISTS clause. Unlike the NOT EXISTS clause, on NOT IN clause nested queries results, there are three optimized hints which are *use_hash*, *merge_aj* and *index_ffs*. However, NOT IN clause nested queries results of correlation INDEX attribute with inner loop foreign table and non-correlated INDEX attribute with non-unique inner SELECT No-Hint approach is still the best solution.

6. Conclusions and future work

The need for a query optimization technique for providing the best response time and the best throughput in query processing is very important. Considering that nested queries are the most complex and expensive operators, this paper has focused on providing an effective optimization technique by using Hints. The chosen technique is precisely a manual tuning that requires users to insert additional comments into an SQL statement.

We have tested various Hints which are: (a) *Optimizer Hints*, (b) *Table join and anti-join Hints*, and (c) *Access method Hints* on our performance evaluations and provided the experiments results. We have applied those Hints to various nested queries such as *IN, ANY/ALL, EXISTS, NOT EXISTS/NOT IN clause nested query*. The overall studies found that adding Hints to a join nested query solely can significantly improve system performance.

In Section 5 we summarized the optimal hint option for different SQL clause and the join structures of each clause. With hint, the performance of IN clause can be reduced up to 20% for disk read and elapsed time. The number of processing time in ANY/ALL clause can be reduced up to 6%. For EXIST clause the number of disk read and the processing time can be reduced up to 37% and 86% respectively. For NOT IN clause, the hint can reduce the number of disk reads and memory used up to 99% and 33% respectively. However, for the NOT EXISTS clause, the hint does not significantly improve the query performance.

Our future work will include evaluating Hints as an effective optimization technique for nested queries using aggregate function [26,28] and object-relational database management system (ORDBMS) queries, since join queries usually include nested queries with a lot of aggregate functions for processing real-world query data that cost complex processing time which may impact on the system's performance. Moreover, join queries in ORDBMS also require a good optimization technique, as the new database design that uses the REF data structure operates differently from the conventional database.

References

[1] M. Andrei, P. Valduriez, User-optimizer communication using abstract plans in Sybase ASE, in: Proceedings of the International Conference on Very Large Databases, Rome, Italy, 2001, pp. 29–38.

[2] D.K. Burleson, Oracle High-Performance SQL Tuning, McGraw Hill, 2001.

[3] S. Chaudhuri, K. Shim, An overview of cost-based optimization of queries with aggregates, IEEE Data Eng. Bull. 18 (3) (1995) 3–9.

[4] S. Chaudhuri, An overview of query optimization in relational systems, in: Proceedings of the ACM Symposium on Principles of Database Systems, Seattle, WA, 1998, pp. 34–43.

[5] S. Chaudhuri, K. Shim, Optimization of queries with user-defined predicates, ACM Trans. Database Syst. 24 (2) (1999) 177–228.

[6] R. Elmasri, S.B. Navathe, Fundamentals of Database Systems, Addison Wesley, 2004.

[7] R.A. Ganski, H.K.T. Wong, Optimization of nested SQL queries revisited, in: Proceedings of the ACM SIGMOD Conference, San Fransisco, CA, 1987, pp. 23–33.

[8] G. Graefe, Query evaluation techniques for large databases, ACM Comput. Surveys 25 (2) (1993) 73–170.

[9] G. Graefe, The cascades framework for query optimization, IEEE Data Eng. Bull. 18 (3) (1995) 19–29.

[10] G. Graefe, W.J. McKenna, The Volcano optimizer generator: extensibility and efficient search, in: Proceedings of the Ninth IEEE International Conference on Data Engineering, Vienna, Austria, 1993, pp. 209–218.

[11] E.P. Harris, K. Ramamohanarao, Join algorithm costs revisited, VLDB J. 5 (1996) 64–84.

[12] M. Jarke, J. Koch, J.W. Schmidt, Introduction to query processing, in: W. Kim, D.S. Reiner, D.S. Batory (Eds.), Query processing in database systems, Springer-Verlag, 1985, pp. 3–28.

[13] N. Kabra, D.J. DeWitt, Efficient mid-query re-optimization of sub-optimal query execution plans, in: Proceedings of the ACM SIGMOD Conference, Seattle, WA, 1998, pp. 106–117.

[14] W. Kim, On optimizing an SQL-like nested query, ACM Trans. Database Syst. 7 (3) (1982) 443–469.

[15] V. Markl, V. Raman, D.E. Simmen, G.M. Lohman, H. Pirahesh, Robust query processing through progressive optimization, in: Proceedings of the ACM SIGMOD Conference, Paris, France, 2004, pp. 659–670.

[16] Oracle, Oracle9i Database Performance Guide and Reference, in: http://www.oracle.com, 2001.

[17] P. Seshadri, J.M. Hellerstein, H. Pirahesh, T.Y.C. Leung, R. Ramakrishnan, D. Srivastava, P.J. Stuckey, S. Sudarshan, Cost based optimization for magic: algebra and implementation, in: Proceedings of the ACM SIGMOD Conference, Montreal, Canada, 1996, pp. 435–446.

[18] A. Swami, Optimization of large join queries: combining heuristics and combinatorial techniques, in: Proceedings of the ACM SIGMOD Conference, Portland, OR, 1989, pp. 367–376.

[19] D. Taniar, J.W. Rahayu, Parallel database sorting, Inform. Sci. 146 (2002) 171–219.

[20] D. Taniar, J.W. Rahayu, Parallel sort–hash object-oriented collection join algorithms for shared-memory machines, Parallel Algorithms Appl. 17 (1) (2002) 85–126.

[21] D. Taniar, J.W. Rahayu, Parallel sort–merge object-oriented collection join algorithms, Int. J. Comput. Syst. Sci. Eng. 17 (3) (2002) 145–158.

[22] D. Taniar, J.W. Rahayu, A taxonomy of indexing schemes for parallel database systems, Distribut. Parallel Databases 12 (1) (2002) 73–106.

[23] D. Taniar, H.Y. Khaw, H.C. Tjioe, J.W. Rahayu, The use of hints in object-relational query query optimization, Int. J. Comput. Syst. Sci. Eng. 19 (6) (2004) 337–346.

[24] D. Taniar, R.B-.N. Tan, C.H.C. Leung, K.H. Liu, Performance analysis of group by-after-join query processing in parallel database systems, Inform. Sci. 168 (2004) 25–50.

[25] L.B. Warshaw, D.P. Miranker, Rule-based query optimization revisited, in: Proceedings of the Eighth ACM International Conference Information and Knowledge Management, Kansas City, MO, 1999, pp. 267–275.

[26] C. Shahabi, M. Jahangiri, D. Sacharidis, Hybrid query and data qodering for fast and progressive range-aggregate query answering, Int. J. Data Warehousing Mining 1 (2) (2005) 49–69.

[27] A.B. Waluyo, B. Srinivasan, D. Taniar, Research in mobile database query optimization and processing, Mobile Information Systems, vol. 1(4), IOS Press, 2005, pp. 225–252.

[28] B. Ozakar, F. Morvan, A. Hameurlain, Mobile join operators for restricted sources, Mobile Information Systems, vol. 1(3), IOS Press, 2005, pp. 167–184.