

The use of hints in object-relational query optimization

David Taniar*, Hui Yee Khaw*, Haorianto Cokrowijoyo Tjioe* and Johanna Wenny Rahayu†

*School of Business Systems, Monash University, Clayton, Victoria 3800, Australia. Email: {David.Taniar,Haorianto.Tjioe}@infotech.monash.edu.au

†Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, Victoria 3803, Australia
Email: wenny@cs.latrobe.edu.au

Object-Relational queries are queries which can handle data objects feature on the existing relational data environment. REF is one of most important data structures in Object-Relational Databases can be explained as a logical pointer to define the link between two tables which similar function to pointers in object oriented programming language. There are three ways of writing REF join queries which are: REF Join, Path Expression Join and Deref/Value Join. In this paper, we study optimization technique for Object-Relational queries using hints. Hints are additional comments which are inserted into an SQL statement for the purpose of instructing or suggesting 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 performance of various object-relational queries particularly in three ways of writing REF queries using the TRACE and TKPROF utilities which provide query execution statistics and execution plans.

Keywords: object-relational query optimization, ORBMS, REF

1. INTRODUCTION

There is a growing need to combine traditional relational data with new object feature into one integrated systems called as object-relational database management system (ORBMS) (Dorsey and Hudicka, 1999; Taniar, Rahayu and Srivastava, 2003). ORBMS is built based on SQL3 model, which consist object data types, row types, collection and abstract data types along with basic relational models (Carey, 1992; Stonebraker and Moore, 1996; Fuh *et al.*, 1999; Taniar, Rahayu and Srivastava, 2003). The advantage of database applying SQL3 model is that the system is more flexible and massive scalability so that the immediate changes in the business process can be accommodated fast and easily (Fuh *et al.*, 1999; Taniar, Rahayu and Srivastava, 2003).

REF is one of essential data structures which have been introduced in Object-Relational Databases serves as a logical

pointer to define the link between two tables (Taniar, Rahayu and Srivastava, 2003). This logical pointer has similar function to pointers in object oriented programming approach. There are three ways of writing REF join queries which are: REF Join, Path Expression Join, and Deref/Value Join. REF Join is the most straightforward method to write Object-Relational queries where the REF data structure is used in the SQL statement to retrieve the REF value (not real data) stored in the ref attribute. Different with REF join, the final result of a Path expression join is the real data taken directly from the referred attribute. Finally, the Deref/Value join method retrieves data from the actual object itself rather than from the table as the other two methods do (Loney and Koch, 2000; Oracle, 2002).

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 (Graefe, 1993; Jarke, 1985). Without any doubt, the benefits resulting from the optimisation of any kind of queries are significant. This will be especially the case when the query being optimized consumes a considerable amount of resources such as physical disk reads and memory buffer.

Generally, in query optimization process, system will automatically choose the most efficient optimization strategy. For example, Oracle is using two kinds of optimizer mode, which are Rule-based Optimizer (RBO) and Cost-based Optimizer (CBO) for its optimization technique (Oracle, 2001). However, this is not enough to serve the best query optimization technique. Since Database Administrator (DBA) often has knowledge about their data than the system optimizer. By using hints on the SQL statement, it will give DBA to manually tune on the SQL statement to improve system performance.

In this paper, we introduce *Object-Relational queries* optimization using hints by focusing on three ways of writing *REF* join queries which are: *REF Join*, *Path Expression Join*, and *DEREF/VALUE Join*. *Object-relational query* is relatively newly concept of SQL3 model (Fortier, 1999; Fuh, et al., 1999; Taniar, Rahayu and Srivastava, 2003). This *object-relational query* is capable in cooperating object data type and traditional relational query. Optimization technique in object-relational query is considered important since this query consumes a considerable amount of resources such as physical disk reads and memory buffers, with hint as an optimization technique could significantly speed up the system performance and reduce the resource needed for processing the object-relational queries.

Basically, hints are some additional comments that are placed within an SQL statement aiming to directly force the optimizer to alter the optimized execution plan (Burleson, 2001). Hints can be categorized into *optimizer hints*, *table join and anti-join hints*, *access method hints* (Burleson, 2001; Oracle, 2001). We will apply these hints on CBO optimizer mode and compare the result with the normal RBO optimizer.

The organisation of the rest of the paper is as follows. In section 2, we discuss about Object-Relational queries, optimization technique using hints and the optimizer modes used in our experiments followed by performance evaluation with some results showing the effectiveness of using hints in section 3. Finally in section 4, we give conclusions of the work and discuss future work.

2. BACKGROUND

In this section we explain the object relational queries, optimization technique using hints and the optimizer modes used in our experiments.

2.1 Object-relational queries

Object-relational queries are using the SQL3 model which has incorporated many new data structures such as *REF* (Fortier, 1999; Fuh et al., 1999; Taniar, Rahayu and Srivastava, 2003). The introduction of the *REF* data structure incorporates the idea of object oriented programming language into database, which means the tables are now viewed as

separate objects (Oracle, 2002; Taniar, Rahayu and Srivastava, 2003). Hence, the *REF* pointer can be used outside the scope of the database. While a foreign key only references the corresponding primary key, the *REF* pointer holds the complete information of the primary table. There is also an association relationship can be applied for *REF* data structure such as many to many, one to many, and one to one (Loney and Koch, 2000). On the other hand, in a conventional DBMS, the idea of the primary key-foreign key is used to maintain data integrity between two tables using the *CONSTRAINT* and *REFERENCES* commands (see Figure 1). That is, data values in the foreign key attribute must be originated from the primary table and these values can only be referred to within the database system.

As can be seen on Figure 2, first we create object with its attributes and then we create another object which has *REF* syntax to previous object, finally we create tables based on those objects. *REF* structure has two important attributes which are: *referred attribute* which stores real data and associated *REF* value and *ref attribute* which stores *REF* value as if it was the real data.

In Figure 3, there are syntaxes of creating table *Student* and *Login*. Attribute *SidLogin* in *Login* table can be identified as *ref attribute*. It stores the pointer value of table *Student* as its real data value. While attribute *StudentId* in *Student* table can be identified as *referred attribute*. It keeps real data value not the pointer value.

We classify Object-Relational query by using *REF* structure mainly based on three ways of writing *REF* join queries which are: *REF Join*, *Path Expression Join*, and *DEREF/VALUE Join* (Taniar, Rahayu and Srivastava, 2003)

2.1.1 REF Join

REF Join is the most straightforward method to write Object-Relational queries where the *REF* data structure is

```
CREATE TABLE <Table1>
(<attribute1> <datatype> PRIMARY KEY,
 <attribute2> <datatype>,
 <.....> <.....>);

CREATE TABLE <Table2>
(<attribute1> <datatype> REFERENCES <Table1> (<attribute1>),
 <attribute2> <datatype>,
 <.....> <.....>);
```

Figure 1 General syntax of creating table on Relational DB

```
CREATE OR REPLACE TYPE <Object1> AS OBJECT
(<attribute1> <datatype>,
 <attribute2> <datatype>,
 <.....> <.....>);

CREATE TABLE <TABLE1> OF <Object1>
(<attribute1> <constraint>
 <attribute2> <constraint>,
 <.....> <.....>);

CREATE OR REPLACE TYPE <Object2> AS OBJECT
(<attribute1> <datatype>,
 <attribute2> <datatype>,
 <.....> <.....>,
 <attribute3> REF <Object1>);

CREATE TABLE <TABLE2> OF <Object2>
(<attribute1> <constraint>
 <attribute2> <constraint>,
 <.....> <.....>,
 <attribute3> SCOPE IS <Object1>);
```

Figure 2 General syntax of creating table on Object-Relational DB

```

CREATE OR REPLACE TYPE Student_Obj AS OBJECT
(StudentID number,
 Student_Surname varchar2(50),
 Email varchar2(50),
 Phone number) /

CREATE TABLE Student OF Student_Obj
(Sid NOT NULL PRIMARY KEY);

CREATE OR REPLACE TYPE Login_Obj AS OBJECT
(LoginID varchar2(20),
 Password varchar2(20) NOT NULL,
 Description varchar2(20),
 SidLogin REF Student) /

CREATE TABLE Login OF Login_Obj
(LoginID NOT NULL PRIMARY KEY,
 SidLogin SCOPE IS Student_Obj) /

INSERT INTO Student VALUES (18811965, 'Julia
Eris', 'eris@monash.edu.au', 967123);

INSERT INTO Login VALUES ('eris', 'my_eris', 'postgrad',
(SELECT REF(a) FROM Student a
WHERE a.Studentid=18811965));

```

Figure 3 Example of implementing REF structure on Object-Relational DB

used in the SQL statement to retrieve the REF value (not real data) stored in the *ref attribute*. Every real data from *referred attribute* always has a unique system generated REF value. REF value is not a real data which the *referred attribute* actually has, it only a pointer address which will point to the real data. Once created a *ref attribute*, the REF value will be save as the real data. The following presents the syntax and a query example for the *REF join structure*:

General Syntax

```

SELECT <Attribute>
FROM <Table1> <alias1>, <Table2> <alias2>
WHERE <alias2>.<Ref Attribute> = REF (<alias1>);

```

Example

```

SELECT s.staffid, s.sname
FROM staff2 s, cust_order2 r
WHERE r.staffid_obj = REF (s);

```

Based on above example, table *cust_order2* consists of *ref attribute* which is *staffid_obj* where it will refer to *referred attribute* in table *staff2*. The REF(s) on that WHERE condition means that all values of all attributes on table *staff2* will be compared with attribute *staffid_obj* on table *cust_order2*.

2.1.2 Path Expression Join

Unlike *REF join*, the final result of a *Path expression join* is the real data taken directly from the referred attribute. Firstly, Oracle optimizer will use the logical pointer (REF value) stored in the *ref attribute* to search the corresponding table for the same value and then retrieve the real data from the *referred attribute*. The following presents the syntax for the path expression join structure:

General Syntax

```

SELECT <Attribute>
FROM <Table1> <alias1>, <Table2> <alias2>
WHERE <alias1>.<Referred Attribute> =
      <alias2>.<Ref Attribute> . <Referred Attribute>;

```

Example

```

SELECT s.staffid, s.sname

```

```

FROM staff2 s, cust_order2 r
WHERE s.staffid = r.staffid_obj.staffid;

```

Based on above example, table *cust_order2* consists of *ref attribute* which is *staffed_obj* where it will refer to *referred attribute* in table *staff2*. WHERE condition on above example, consist of a comparison between *staffid* attribute in table *staff2* with *staffid_obj* attribute as a ref attribute in table *cust_order2* where a specific attribute on *staffid_obj* attribute is *staffid*.

2.1.3 Deref/Value Join

The *DEREF/VALUE join* method retrieves data from the actual object itself rather than from the table as the other two methods do (Loney and Koch, 2000; Oracle, 2002). The direct access to objects limits the competence of the method from certain operations.

General Syntax

```

SELECT <Attribute>
FROM <Table1> <alias1>, <Table2> <alias2>
WHERE DEREF (<alias2> . <Ref Attribute>) = VALUE
(<alias1>);

```

Example

```

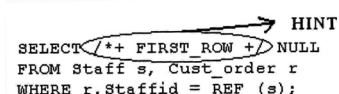
SELECT s.staffid, s.sname
FROM staff2 s, cust_order2 r
WHERE DEREF (r.staffid_obj) = VALUE(s);

```

Based on above example, table *cust_order2* consists of *ref attribute* which is *staffid_obj* as an object which as the same information on object *staff2*. Using *DEREF/VALUE* we look at the data as an object, thus on above WHERE condition DEREF syntax contain *staffid_obj* object which compared values of a ref attribute from table *cust_order2* with values on object table *staff2* using syntax VALUE(s).

2.2 Optimization technique: Hints

As can be seen in Figure 4, all hints have to be written after a SELECT /UPDATE /INSERT/ DELETE statement with /*+ hint */. In this example, we use *first_row* hint to optimize the Object-Relational queries between tables *Staff* and *Cust_order*. Oracle provides an extensive list of hints that can be used to tune various aspects of a query performance (Oracle, 2001). We categorize hints for Object-Relational queries optimization into three parts: (a) Optimizer Hints, (b) Table Join and Anti-Join Hints, (c) Access Method Hints (Burleson, 2001; Oracle, 2001). *Optimizer hints* apply different optimizer to a query. Meanwhile, *table join and anti-join hint* focus on applying different hints to join operations. *Access method hints* concern with using index and table access hint for optimizing Object-Relational queries operation based on its access path.



```

SELECT /*+ FIRST ROW */ NULL
FROM Staff s, Cust_order r
WHERE r.Staffid = REF (s);

```

Figure 4 Using a hint in an SQL statement

We categorize hints for Object-Relational queries optimization into three parts: (1) Optimizer Hints, (2) Table Join and Anti-Join Hints, (3) Access Method Hints. The details of them are explained in the following sections.

2.2.1 Optimizer Hints

Optimizer hints apply a different optimizer to a query, which in turns redirect the overall processing goal (Burlleson, 2001). Oracle offers four kinds of optimizer hints, which are *all_rows*, *first_rows*, *rule*, and *choose* (refer to Figure 5). The *all_rows* hint explicitly chooses the cost-based approach (CBO) 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 rule-based (RBO) and cost-based (CBO) approaches for a 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-based approach and optimizes with the goal of best throughput. If the data dictionary does not have statistics for these tables, then it uses the rule-based approach.

The *first_rows* hint favors full-index scan and invokes the cost-based optimizer (CBO) to return the first row of a query within the shortest processing time and minimum resource usage. Since its goal is to achieve the best response time, it is suitable used for Online Transaction Processing (OLTP). In our performance evaluation, only the *first_rows* hint is used due to the following considerations:

- The default optimizer mode is set to rule-based (RBO). Thus, the *rule* hint is inapplicable or unnecessary.
- The adding of hints in a rule-based environment will automatically redirect the optimizer to the *all_rows* mode.
- The *choose* hint does not have a fixed goal, as it causes the optimizer to choose between the rule-based and the cost-based approach.

Figure 6 shows that the optimizer goal has been changed from no hint to add *first_rows* hint to the select statement. In Figure 6(a) the goal of select statement is rule (rule based) in which we do not apply any hint since this rule hint is a default optimizer mode from the system. By applying *first_rows* hint in Figure 6(b), the select statement goal is forced to change from its default. Although the execution plan between Figure 6(a) and 6(b) is quite the same, however by applying a hint, it gives a better performance result.

2.2.2 Table Join and Anti-Join Hints

There are several hints on table join and anti-join hints pro-

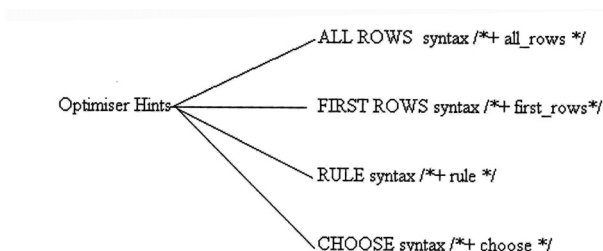


Figure 5 Optimizer hints

<pre> select null from staff2 s, emp_order? c where dect(c.staffid) = value(s) </pre>	<pre> select /*+ first_rows */ null from emp_order? c, staff2 s where dect(c.staffid) = value(s) </pre>
<p>Rows Execution Plan</p> <pre> 0 SELECT STATEMENT GOAL: RULE 1015 NESTED LOOPS 1016 NESTED LOOPS (OUTER) 1016 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'EMP_ORDER?' 1015 TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'STAFF2' 2030 INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'SYS_C00831' (UNIQUE) 1015 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'STAFF2' </pre>	<p>Rows Execution Plan</p> <pre> 0 SELECT STATEMENT GOAL: HINT: FIRST_ROWS 1015 NESTED LOOPS 1016 NESTED LOOPS (OUTER) 1016 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'EMP_ORDER?' 1015 TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'STAFF2' 2030 INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'SYS_C00831' (UNIQUE) 1015 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'STAFF2' </pre>
(a) No Hint (Rule-based)	(b) Hint -first_rows (Cost-based)

Figure 6 The effect of adding optimizer hint

vided by Oracle as shown in Figure 7, 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 source with a nested loop join uses the specified table as the inner table. In general, the default join method in Oracle is the nested loop join. Therefore, *use_nl* hint is not used in the performance evaluation. There are two hints which can be used for Object-Relational queries 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.
- The *use_merge* hint. Alternatively, the hint forces the optimizer to join tables using the sort-merge operation.

As can be seen from Figure 8, either *use_merge* example and *use_hash* example, it has been specified tables student and login to be joined to the row source resulting from joining the previous tables in the join order either using a sort merge join and a hash join.

2.2.3 Access Method Hints

Access method hints can be classified into *hints without index* and *hints with index*. *Rowid* hint is one of *hints without index*. *Hints with index* consist of *index* hint, *index_join* hint and *index_ffs* hint. As shown in Figure 9, there is a select statement with *index_ffs* hint consist of two parameters which are *student* as the table *student* and *sid_index* as the table index of table *student*. We will briefly explain each hint on access method hints as follow:

- The *Rowid* hint. The hint forces the optimizer to scan the specified table by rowid, which is generally gathered from

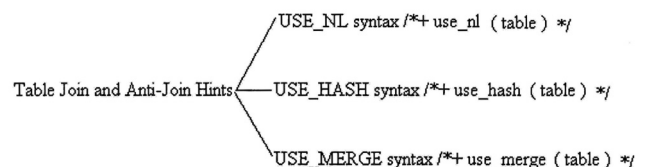


Figure 7 Table Join and Anti-Join hints

<pre> SELECT p.student_surname, l.password /*+USE_MERGE(student login)*/ FROM student p, login l WHERE l.sidlogin = REF(p) And l.sidlogin.studentid = 12345; </pre>	<pre> SELECT p.student_surname, l.password /*+USE_HASH(student login)*/ FROM student p, login l WHERE l.sidlogin = REF(p) And l.sidlogin.studentid = 12345; </pre>
USE_MERGE HINT	USE_HASH HINT

Figure 8 Table Join and Anti-Join Hints examples

```

SELECT p.student_surname, l.password
  /*INDEX_FFS(student sid_index)*/
FROM student p, login l
WHERE l.sidlogin = REF(p)
And l.sidlogin.studentid = 12345;

```

Figure 9 Syntax and example of access method hints

an index. Syntax `/*+ROWID(<table>)*/`

- The *Index* hint. The hint explicitly instructs the optimizer to use an index scan as the access path to the specified table. Syntax `/*+INDEX(<table> <index name>)*/`
- The *Index_join* hint. The hint alters the optimizer to access the specified table using the index join method. Syntax `/*+INDEX_JOIN(<table> <index name>)*/`
- 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 unnecessary to access the table rows. Syntax `/*+INDEX_FFS(<table> <index name>)*/`

2.3 Optimizer modes

There are two kinds of optimizer modes in Oracle, which are: *rule based optimizer* (RBO) and *cost based optimizer* (CBO) (Oracle, 2001). RBO is used as a default optimizer mode in Oracle where no hint is applied in this optimizer mode. Different from RBO, in CBO all hint regards its category will be used in this optimizer mode. We use RBO to compare the advantages of using hints in CBO to optimize join nested query operations.

The *Rule-Based optimizer* (RBO) mode is using a heuristic optimization technique. Its 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 the most restrictive operations (Elmasri and Navathe, 2000). Apparently, the re-ordering of the leaf nodes on the query tree can have significant impact on the system performance and hence good heuristics should be applied whenever possible. However, as shown on performance evaluation section, this optimizer mode is less effective on many join nested queries optimization if compared with using hints on CBO optimizer mode.

The *Cost-Based Optimizer* (CBO) mode determines which execution plan is 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 (Harris and Ramamohanarao, 1996). 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 looking at the statistics.

CBO 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 CBO is more flexible and intelligent in choosing the SQL operations. Therefore, it should make a better execution plan than the RBO does if it has all the statistical information it needs. Moreover, CBO has two main optimization goals, which are:

the fastest response time with minimum resource usage and the best throughput with minimum total resource consumption.

Unfortunately, CBO's performance is unpredictable and not as stable as the RBO. The introduction of hints in cost-based optimization has indicated the inability of the CBO to produce the best strategy for every query. By applying hints as an optimization technique in CBO, it will improve a significant result in faster response time and lower resource consumption.

3. OBJECT-RELATIONAL QUERIES OPTIMIZATION

As been discussed above, Object-Relational queries are query which either can be applied by combining traditional relational database with new object feature or just using only objects data. Since object-relational queries is different with a normal relational join queries and also this query consumes a considerable amount of resources such as physical disk reads and memory buffers. REF is one of most important data structure in Object-Relational database has three ways of writing REF join query which are *REF Join*, *Path Expression Join*, *DEREF/VALUE Join* (Taniar, Rahayu and Srivastava, 2003). Here we use hint as an optimization technique to speed up the time processing, reduce the physical disk reads and memory buffers. In this section, we present our performance evaluation result incorporating hints with three ways of writing REF join query using TKPROF utility in Oracle.

3.1 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 (Järke *et al*, 1985). 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 on the performance of the query processing. For instance, as show on the following illustration (see Figure 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* but 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, which are:

```

select null
from staff2 s , cust_order2 r
where r.staffid = ref(s)
Rows      Execution Plan
-----
0  SELECT STATEMENT    GOAL: RULE
1015  NESTED LOOPS
1016    TABLE ACCESS    GOAL: ANALYZED (FULL) OF 'CUST_ORDER2'
1015      INDEX          GOAL: ANALYZED (UNIQUE SCAN) OF 'SYS_C00831' (UNIQUE)

```

Figure 10 Example of execution plan

- Step 1: Nested Loops
 Step 2: Full table access to *Cust_order2* table
 Step 3: Index unique scan on the *Staff2_Staffid_Pk* or *SYS_C00831* of the *Staff2* table

The operations below the nested loops are indented indicating that they should be executed prior to the join operation. Hence, the first execution is Step 2, which will retrieve a row from the *Cust_order2* table and return it to Step 1. Then, Step 1 will send the *Staffid* (Staff number) to Step 3 for every row obtained from Step 2. In Step 3, the optimizer will perform an index unique scan on the *Staff2_Staffid_Pk* or *SYS_C00831* of the *Staff2* table. Finally, Step 3 will return the row to Step 1, where the row will be joined with that from step2 if the *staffid* equal to *REF(s)*. 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 only be activated by setting the relevant parameters. There are two types of parameters that need to be carefully set (Burleson, 2001): (a) file size and location parameters (eq. `User_dump_dest = c:\orahome\trace-file`) (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 (Burleson, 2001). 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:

```
Tkprof <TraceFile> <OutputFile> [explain
= <username>/<password>]
```

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 on Figure 11, TKPROF result is divided into three parts which are: *sql statement*, *statistic information* and *execution plan*. On 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 on the phases and the statistics measurements.

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 Figure 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.

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 statistic, 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.

3.2 Experimental results

For the purpose of experimentation in the paper, a Sales database has been created as a sample database. Table 1 shows the details of the database, while Figure 12 depicts the relationships between each individual table along with SQL for creating Object-Relational tables.

The default optimization approach used in the experiments is set to RBO, which uses the heuristic rules to optimize a given query. The approach is chosen to replace the default mode due to its stability and predictability as stated by Burleson (2001). Moreover, the adding of hints to the SQL statement will redirect the optimization approach to cost-based (CBO). Hence, analysis can be carried out on the

Table 1 Sales database

Table	Attributes	Records
Customer2	(custid, cname, addr, city)	10,010
Staff2	(staffid, sname, city)	1,010
Cust_order2	(ordered, orderdate, custid, staffed)	10,000
Item	(itemid, itemdesc, category)	999
Inventory	(invid, itemid, curr_price, qoh)	9500
Orderline	(ordered, invid, order_price, qty)	15,000

```
select null
from staff2 s , cust_order2 r
where r.staffid = ref(s)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	1.36	96	507	3	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	68	0.00	0.04	46	167	4	1015
total	70	0.00	1.40	142	674	7	1015

Rows	Execution Plan
0	SELECT STATEMENT GOAL: RULE
1015	NESTED LOOPS
1016	TABLE ACCESS GOAL: ANALYZED (FULL) OF 'CUST_ORDER2'
1015	INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'SYS_C00831' (UNIQUE)

Figure 11 Example of TKPROF result (formatted trace report)

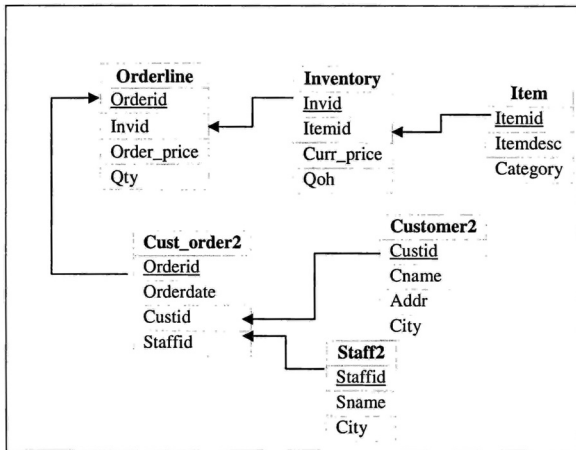


Figure 12 Sales order systems

performance of the RBO and the CBO in the later stage to reveal the best approach for optimizing a given query. In fact, the experiment results will enable comparison to be made on three types of optimization approaches, which are: (a) Rule-based approach (by default), (b) Cost-based approach with a goal of *best response time* (*first_rows* mode), (3) Cost-based approach with a goal of *best throughput* (*all_rows* mode).

Although the *all_rows* hint is not used in the experiments, the goal of the best throughput can still be accomplished. According to Burleson (2001), the actual optimizer mode will be the *all_rows* mode whenever a hint (except the *rule* hint) is added even under the rule-based mode. Therefore, results produced under different approach are obtained by the following mean:

- Execute query without adding hints – Query is optimized with heuristic rules by the RBO.
- Add the optimizer hint (*first_rows*) – Query is optimized by the CBO to achieve the *best response time*.
- Add the other hints (join operation and access method hints) – Query is optimized by the CBO to achieve the *best throughput*.

In the next section we have experimental hints and its results based on the Object-Relational queries on section 2.

```

CREATE OR REPLACE TYPE CUST_OBJ AS OBJECT
(CUSTID CHAR (5),
CNAME  VARCHAR2 (15),
ADDR   VARCHAR2 (30),
CITY   VARCHAR2 (15));

```

```

CREATE TABLE CUSTOMER2 OF CUST_OBJ
(CUSTID NOT NULL PRIMARY KEY);

```

```

CREATE OR REPLACE TYPE STAFF_OBJ AS OBJECT
(STAFFID CHAR (5),
SNAME  VARCHAR2 (15),
CITY   VARCHAR2 (15));

```

```

CREATE TABLE STAFF2 OF STAFF_OBJ
(STAFFID NOT NULL PRIMARY KEY);

```

```

CREATE OR REPLACE TYPE CUST_ORDER_OBJ AS OBJECT
(ORDERID      CHAR(6),
ORDERDATE     DATE,
CUSTID        REF CUST_OBJ,
STAFFID       REF STAFF_OBJ);

```

```

CREATE TABLE CUST_ORDER2 OF CUST_ORDER_OBJ
(ORDERID NOT NULL PRIMARY KEY,
CUSTID   SCOPE IS CUSTOMER2,
STAFFID  SCOPE IS STAFF2);

```

3.2.1 Object-relational query optimization results with REF join

After making the comparison between the other hints (see Figure 13), it is concluded that the *index_ffs* hint is still the best hint that produces the fastest execution plan with the lowest physical disk reads for Object-Relational queries with *REF Join*. The execution plan in Figure 14(b) shows that the hint causes the optimiser to use the hash join operation in sync with an index fast full scan on the indexed attribute (Staff2_Staffid_Pk or SYS_C00831). As a result, the statistics show a dramatic decrease in both the processing time and the number of disk reads.

While it was expected that the number of memory reads would increase due to the building of the hash table, it has actually decreased by about 85%. After analysing the TKPROF result, it is observed that the extensive memory reads by the RBO (Figure 14(a)) is likely to be caused by the inefficient nested loop join algorithm where unnecessary scanning is carried out on the inner table that has an indexed

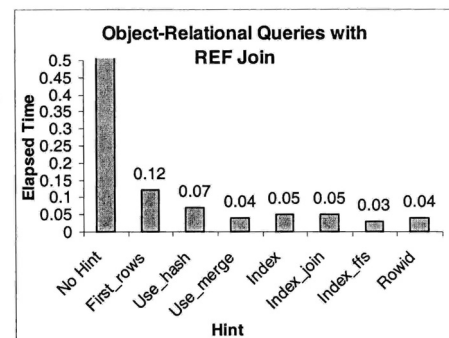


Figure 13 Comparison of all hints experimented on Object-Relational queries with REF Join

<pre>select null from staff s, cust_order2 c where c.staffid = ref(s)</pre>								<pre>select /*+ index_ffs(s) */ null from cust_order2 c, staff s where c.staffid = ref(s)</pre>							
call	count	cpu	elapsed	disk	query	current	rows	call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	1.36	96	507	3	0	Parse	1	0.00	0.01	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0	Execute	1	0.00	0.00	0	0	0	0
Fetch	68	0.00	0.04	46	167	4	1015	Fetch	68	0.00	0.02	30	99	8	1015
Total	70	0.00	1.40	142	674	7	1015	Total	70	0.00	0.03	30	99	8	1015
Rows Execution Plan								Rows Execution Plan							
0 SELECT STATEMENT GOAL: RULE								0 SELECT STATEMENT GOAL: RULE							
1015 NESTED LOOPS								1015 HASH JOIN							
1016 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'CUST_ORDER2'								10 INDEX GOAL: ANALYZED (FAST FULL SCAN) OF 'SYS_C00831' (UNIQUE)							
1015 INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'SYS_C00831' (UNIQUE)								1015 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'CUST_ORDER2'							

(a) No Hint (Rule-based)

(b) The Best Hint – index_join (Cost-based)

Figure 14 Comparison of rule-based and hint optimised trace reports for the Object-Relational queries with REF Join

join attribute (SYS_C00831). This can be seen from the number of rows processed in the execution plan produced under the rule-based optimization. While the inner table (*Staff2*) only has 10 records, it has to be accessed as many times as the number of records contained in the outer loop (1,015), which is a hundred times more than the necessary scan. This is because all records in the outer table match with those of the inner. For these reasons, it is apparent that the hash join algorithm is a far more efficient solution for the processing of this type of Object-Relational queries.

3.2.2 Object-relational query optimization results with path expression join

Once again, the *index_ffs* hint is expected to have the most outstanding performance among the other hints on Object-Relational queries with *Path Expression Join*. Although the *rowid* hint is able to process the given query within 0.03 seconds as well (see Figure 15), it consumes more resources in terms of both physical disk and memory reads. Therefore, the *index_ffs* hint is chosen to be the best hint for this path expression join. As usual, the hint has invoked the hash join operation to replace the original nested loop join leading to a remarkable cost reduction. These costs include the processing time, the number of data blocks retrieved from the physical disk, and the memory reads (see Figure 16).

3.2.3 Object-relational query optimization results with DEREf/VALUE Join

While it was expecting that the *index_ffs* hint would once

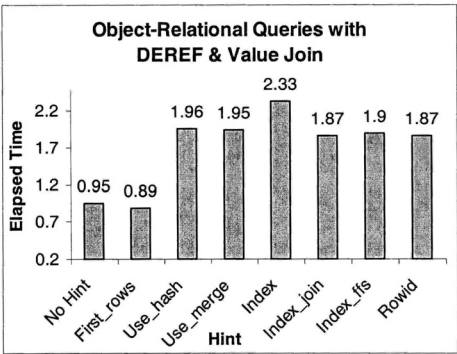


Figure 17 Comparison of all hints experimented on Object-Relational queries with DEREf/VALUE Join

again produce the best execution plan for Object-Relational queries with *DEREF/VALUE Join*, the *first_rows* hint turns out to be the outstanding one (see Figure 17). This means, the CBO alone is able to process the given query efficiently without the intervention of any hints. However, the improvement is not as significant. As can be seen on Figure 18, both the RBO and the CBO produce an identical execution plan, the interesting part is that the plan produced by the CBO is 0.06 seconds faster than that of the RBO, and the number of physical disk reads is slightly reduced (2 data blocks). Therefore, the DBA has two options when encounter the processing of this join, which are: set the optimiser mode as rule-based and then add the *first_rows* hint to the SQL statement, or set the optimiser mode as cost-based with the goal of achieving the best response time (*first_rows*).

3.3 Discussions

There can be more than one goal when it comes to optimize 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 maybe at the expense of resources or overall processing time. On the other hand, tasks such as generating a large report generally require the

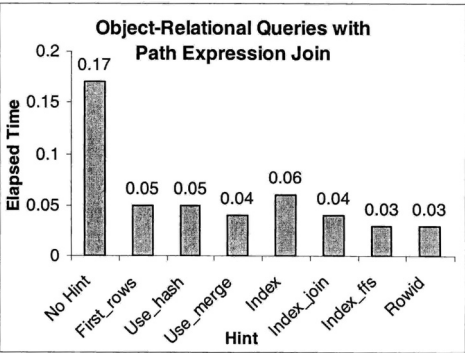


Figure 15 Comparison of all hints experimented on Object-Relational queries with Path Expression Join

select null from staff s, cust_order c where s.staffid = c.staffid							
call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.05	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	68	0.00	0.12	44	2197	4	1015
Total	70	0.00	0.17	44	2197	4	1015
Rows Execution Plan							
0 SELECT STATEMENT GOAL: RULE							
1015 NESTED LOOPS							
1016 NESTED LOOPS (OUTER)							
1016 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'CUST_ORDER'							
1015 TABLE ACCESS GOAL: ANALYZED (BY ROWID) OF 'STAFF'							
2000 INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'SYS_C00831' (UNIQUE)							
1015 INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'SYS_C00830' (UNIQUE)							

(a) No Hint (Rule-based)

select /*+ index_ffs(s) */ null from staff s, cust_order c where s.staffid = c.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	68	0.00	0.02	35	55	12	1015
Total	70	0.00	0.03	35	55	12	1015
Rows Execution Plan							
0 SELECT STATEMENT GOAL: RULE							
1015 HASH JOIN							
10 INDEX GOAL: ANALYZED (FAST FULL SCAN) OF 'SYS_C00830'							
1015 HASH JOIN (OUTER)							
1015 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'CUST_ORDER'							
10 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'STAFF'							

(b) The Best Hint - *index_ffs* (Cost-based)

Figure 16 Comparison of rule-based and hint optimized reports for Object-Relational queries with Path Expression Join

select null from staff s, cust_order c where deref(r.staffid) = value(s)								select /*+ first_rows */ null from cust_order c, staff s where deref(r.staffid) = value(s)							
call	count	cpu	elapsed	disk	query	current	rows	call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.04	0	0	0	0	Parse	1	0.00	0.02	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0	Execute	1	0.00	0.00	0	0	0	0
Fetch	68	0.00	0.91	29	2164	4064	1015	Fetch	68	0.00	0.87	27	2164	4064	1015
total	70	0.00	0.95	29	2164	4064	1015	total	70	0.00	0.89	27	2164	4064	1015
Rows Execution Plan								Rows Execution Plan							
0 SELECT STATEMENT GOAL: FULL								0 SELECT STATEMENT GOAL: HINT: FIRST_ROWS							
1015 NESTED LOOPS								1015 NESTED LOOPS							
1016 NESTED LOOPS (OUTER)								1016 NESTED LOOPS (OUTER)							
1016 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'CUST_ORDER'								1016 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'CUST_ORDER'							
1015 TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'STAFF'								1015 TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'STAFF'							
2000 INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'SYS_C00831' (UNIQUE)								2000 INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'SYS_C00831' (UNIQUE)							
1015 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'STAFF'								1015 TABLE ACCESS GOAL: ANALYZED (FULL) OF 'STAFF'							

(a) No Hint (Rule-based)

(b) The Best Hint – first_rows (Cost-based)

Figure 18 Comparison of rule-based and hint optimised trace reports for the Object-Relational queries with DEREf & VALUE Join

Table 2 Optimal hint(s) for each join structure

Join Structure	Optimal Hint(s)
REF Join	Index_ffs
Path Expression Join	Index_ffs
DEREF & VALUE Join	First_rows

best throughput (minimum total resource consumption) 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 2 recommends the hint that is best for each individual join structure.

On Object-Relational queries results either with *REF Join* and *Path Expression Join*, it has found that *index_ffs* hint is superior among other hints. Different with *index_ffs* hint, *first_rows* hint is the best hint on Object-Relational queries with *DEREF/VALUE Join* results.

4. CONCLUSIONS AND FUTURE WORKS

The need for query optimization technique on providing the best response time and the best throughput in query processing is very important. Considering join queries as 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 on Object-Relational queries. The overall studies found that adding hints to Object-Relational queries on various ways of writing REF join queries such as *REF Join*, *Path Expression Join*, and *DEREF/VALUE Join* solely can significantly improve system performance.

On our future works include evaluating hints as an effective optimization technique on Object-Relational with aggregation queries and inheritance queries. Since those types of queries usually represent the processing of real-world query

data which have more complex processes and higher processing time which may impact to the system performance.

REFERENCES

- 1 **Burleson, D.K.** *Oracle High-Performance SQL Tuning* (1th ed.): McGraw Hill, 2001.
- 2 **Carey M. et al.** O-R, What Have They Done to DB2? *Proceedings of the Very Large Database International Conference (VLDB)*, 1999.
- 3 **Date C.J.** *An Introduction to Database Systems* (7th ed.): Addison-Wesley, 2000.
- 4 **Dorsey, P. and Hudicka, J.R.** *Oracle 8 Design Using UML Object Modeling*, Oracle Press, McGraw Hill, 1999.
- 5 **Elmasri, R., and Navathe, S.B.** *Fundamentals of Database Systems* (3rd ed.) Sydney: Addison Wesley, 2000.
- 6 **Fortier, P.J.** *SQL3 Implementing the SQL Foundation Standard*, McGraw Hill, 1999.
- 7 **Fuh Y-C et al.** Implementation of SQL3 Structured Types with Inheritance and Value Substitutability, *Proceedings of the Very Large Databases International Conference (VLDB)*, 1999.
- 8 **Graefe, G.** Query Evaluation Techniques for Large Databases, *ACM Computing Surveys*, vol. 25, no. 2, pp. 73–170, June 1993.
- 9 **Graefe, G. and McKenna, W.J.** The Volcano optimizer generator: Extensibility and efficient search, *In Proceedings of the IEEE Conf. on Data Engineering. IEEE*, New York 1993.
- 10 **Harris, E.P. and Ramamohanarao, K.** Join Algorithm Costs Revisited, *The VLDB Journal*, vol. 5, pp. 64–84, 1996.
- 11 **Jarke, M. et al.** Introduction to Query Processing, *Query Processing in Database Systems*, W.Kim et al. (eds.), Springer-Verlag, pp. 3–28, 1985.
- 12 **Loney, K., and Koch, G.** *Oracle 8i: The Complete Reference*, Osborne McGraw-Hill, 2000.
- 13 **Loney, K., and Koch, G.** *Oracle 9i: The Complete Reference*, Oracle Press, 2002.
- 14 **Mishra, P. and Eich, M.H.** Join Processing in Relational Databases, *ACM Computing Surveys*, vol. 24, no. 1, pp.63–113, March 1992.
- 15 **Oracle** Oracle9i Database Performance Guide and Reference, <http://www.oracle.com>, 2001.
- 16 **Oracle** Oracle9i Database Concepts, <http://www.oracle.com>, 2002.
- 17 **Ramakrishnan, R. and Gehrke, J.** *Database Management Systems* (2nd ed.): McGraw Hill, 2000.
- 18 **Stonebraker, M. and Moore, D.** *Object Relational DBMS's: The Next Great Wave*, Morgan Kaufmann, 1996.
- 19 **Taniar, D., Rahayu, J.W. and Srivastava, P.G.** Chapter 12: A Taxonomy of Object-Relational Queries, *Effective Databases for Text & Document Management*, S.A. Becker(ed.), pp. 183–220, IRM Press, 2003.

