PINNACLE

Oracle Professional

Solutions for High-End
Oracle® DBAs and Developers

Avoid Costly Joins with FBIs

Pedro Bizarro

DOWNLOAD

In this article, Pedro Bizarro describes how to use Function-Based Indexes to avoid costly joins. Data warehouses are particularly suited to the usage of this technique, since joins tend to be heavy operations and dimensions are scarcely updated. This technique is only applicable in Oracle8i.

WITH Function-Based Indexes (FBIs), I was able to alter an execution plan from a nested loop that spread over five tables, down to bitmap index scans and ROWID accesses in one (!) table. The original query was reduced from 809 seconds to 123 seconds.

You've probably read that FBIs allow you to use indexes even in WHERE clauses like this:

```
SELECT …
  FROM …
 WHERE upper(name) = …;

or
  …
 WHERE extract_year(date) = …;
```

In a sense, an FBI is a bit like a materialized view. Some of the work is already done and stored before running the query. FBIs are especially useful when you use a function as a search condition. Without FBIs, the function is

September 2000

Volume 7, Number 9

1 Avoid Costly Joins with FBIs
Pedro Bizarro

8 A Basic Implementation of Roles
Sumathy Thankam Panicker

10 DBMS_OUTPUT.PUT_LINE Replacement, Caching USER, and Displaying the Call Stack
Steven Feuerstein

14 Log Switch Monitoring
Dan Hotka

17 Tip: Consider Building Oracle Forms Based on Database Views
Roy F. Gomes

18 Tip: Views with Sorts in Oracle8i
Jay Wortman

20 September 2000 Source Code

DOWNLOAD

Indicates accompanying files are available online at http://www.oracleprofessionalnewsletter.com.

executed at runtime, and for every run of the query and every record, the function is executed once (or maybe more, depending on the search). If you use FBIs, the function's results are stored in an index before running the queries. All that's left to do at runtime is to search for the value in the B-tree or in the bitmap. The function wouldn't be executed any more.

Deterministic function

The function used in an FBI may be user-defined but must be deterministic—that is, two executions must return the same value. To ensure that different executions will always return the same value, a deterministic function must not reference package variables or the database. You can declare such a function using the DETERMINISTIC keyword:

```
CREATE OR REPLACE FUNCTION my_function (…)
                                      RETURN …
                                      DETERMINISTIC
                                      IS

BEGIN
    [function code]
    RETURN …;
END my_function;
```

Oracle8i SQL Reference (page 7-270) states, "DETERMINISTIC is an optimization hint that allows the system to use a saved copy of the function's return result (if such a copy is available). The saved copy could come from a materialized view, a function-based index, or a redundant call to the same function in the same SQL statement. The query optimizer can choose whether to use the saved copy or re-call the function. The function should reliably return the same result value whenever it is called with the same values for its arguments. Therefore, do not define the function to use package variables or to access the database in any way that might affect the function's return result, because the results of doing so will not be captured if the system chooses not to call the function."

Can you imagine the power you could harness from FBIs if they could read from the database? Imagine that the base function returns data from other tables and stores it as index data. Remember the scott/tiger schema from Oracle. The first query that follows is a normal join

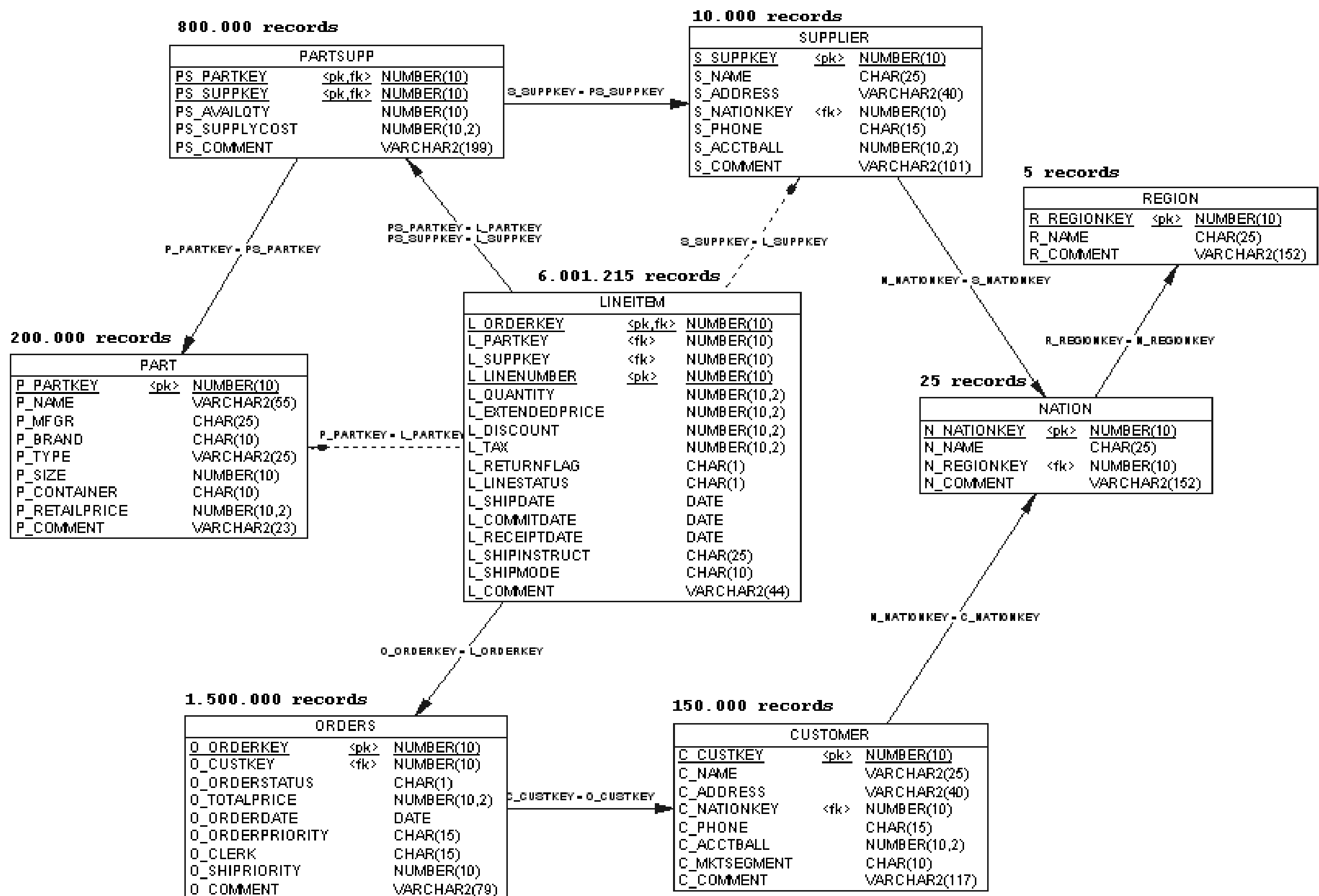

Figure 1. The TPC-H schema with load data required for a scale factor 1 database.

between the EMP and DEPT tables. But the second query reads only from a table and an index. Yet, it produces the same result:

```
/* Retrieves all employees whose department name   */
/* is same value.                                   */
SELECT e.name
  FROM emp e,                /* Retrieves data     */
       dept d                /* from 2 tables.     */
 WHERE e.deptno = d.deptno /* Join condition.      */
   AND d.name = …;           /* Some condition.    */


SELECT e.name                        /* One table,  */
  FROM emp e                         /* no joins.   */
 WHERE get_dept_name(deptno) = …;   /* Function    */
                                     /* used in     */
                                     /* the FBI     */
                                     /* and stored  */
                                     /* in an index */
```

You're probably uncomfortable with the suggestion of having a deterministic function reading the database—I'll address that later in the article.

Setting the scene

My motivation to use FBIs appeared to me when I was benchmarking with TPC-H. As you probably know, TPC-H is a set of tables, data, and queries to benchmark software and hardware that implements a data warehouse. Although the TPC-H schema isn't a perfect example (with dimensions and facts neatly arranged), it's widely accepted that it mimics both the fundamental DW properties as well as the real world idiosyncrasies (see Figure 1 on page 2).

In Figure 1, the solid arrows represent the foreign key

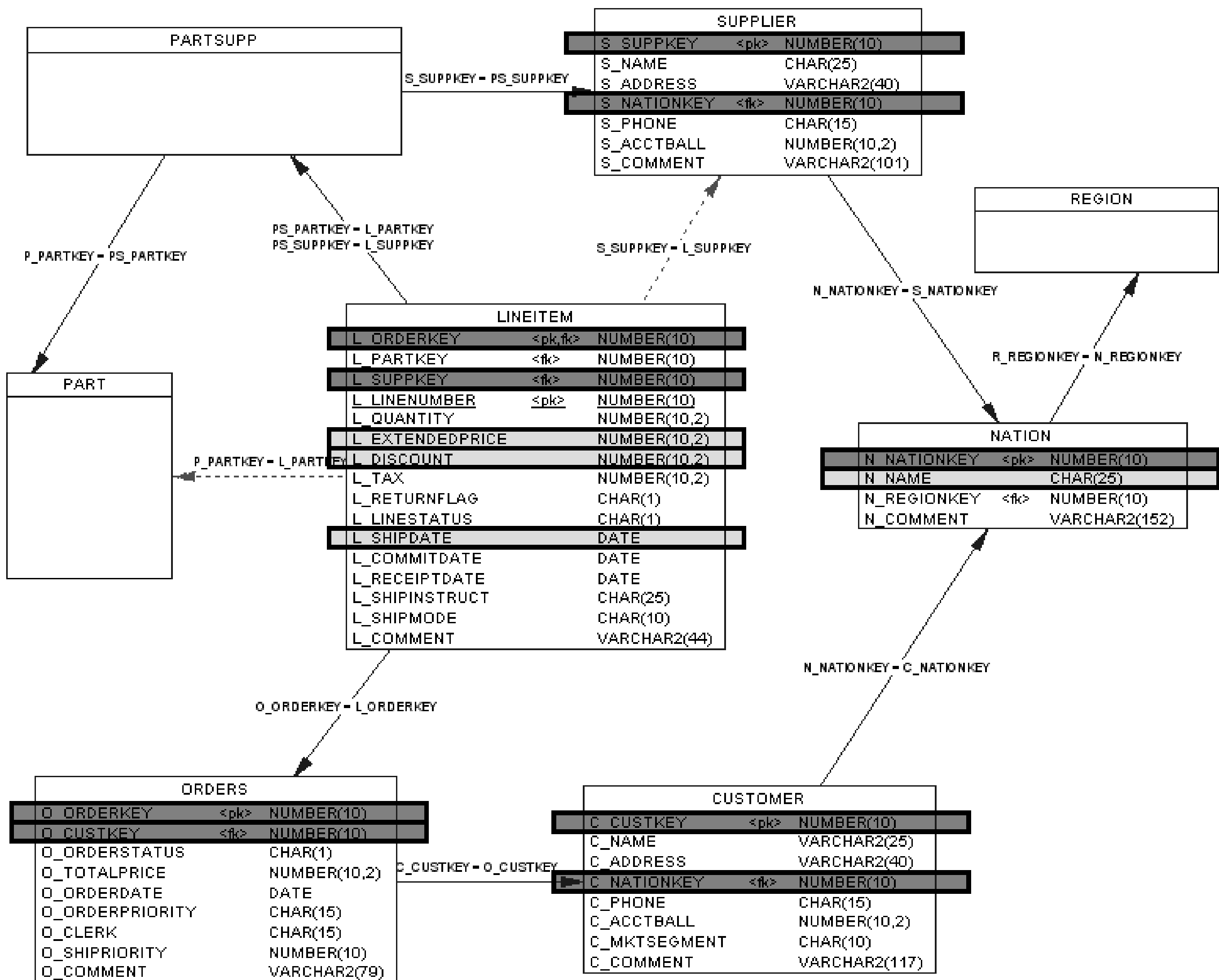

Figure 2. The TPC-H query number 7 visual representation.

(FK) relations. The dotted-line-arrows aren't real FKs in the sense that they don't exist in the schema data dictionary of constraints. But I can assume that they're normal FKs due to the relations of LINEITEM to PARTSUPP, and PARTSUPP to both PART and SUPPLIER.

TPC-H query number 7 is called Volume Shipping. It finds, for two given Nations, the gross discount revenues derived from Lineitems in which Parts where shipped from a Supplier in either Nation to a Customer in the other Nation during 1995 and 1996. Two nations are given as input parameters. The full query 7 text is shown in Listing 1.

Listing 1. TPC-H query number 7—Volume Shipping.

```
SELECT
  supp_nation,
  cust_nation,
  l_year,
  sum(volume) revenue
FROM
  (
    SELECT
      n1.n_name supp_nation,
      n2.n_name cust_nation,
      to_char(l_shipdate, 'YYYY') l_year,
      l_extendedprice * (1 - l_discount) volume
    FROM
      supplier,
      lineitem,
      orders,
      customer,
      nation n1,
      nation n2
    WHERE
          s_suppkey = l_suppkey
      AND o_orderkey = l_orderkey
      AND c_custkey = o_custkey
      AND s_nationkey = n1.n_nationkey
      AND c_nationkey = n2.n_nationkey
      AND (
              (    n1.n_name = 'FRANCE'
               AND n2.n_name = 'GERMANY')
           OR (    n1.n_name = 'GERMANY'
               AND n2.n_name = 'FRANCE'))
      AND l_shipdate
          BETWEEN to_date('1995-01-01', 'yyyy-mm-dd')
              AND to_date('1996-12-31', 'yyyy-mm-dd')
  ) shipping
GROUP BY
  supp_nation,
  cust_nation,
  l_year
ORDER BY
  supp_nation,
  cust_nation,
  l_year;
```

Since this might be a bit of a dense query to grasp, I'll provide a visual representation for it (see Figure 2 on page 3).

In Figure 2, the dark gray columns represent values read just to do the joins between tables. Light gray columns represent the columns I really have to read, either as input or output values. Note that I have to join LINEITEM to ORDER, and then to CUSTOMER, and then to NATION in order to find the Customer's Nation. Likewise, I have to join LINEITEM to SUPPLIER and NATION to find the Supplier's nation.

The problem with query 7 is that to get from LINEITEM to NATION, I have to go through ORDERS (1,500,000 records) and CUSTOMER (150,000 records). That will be a pretty heavy nested loop.

Wouldn't it be nice to skip ORDERS, CUSTOMER, and SUPPLIER? I could, if I changed the schema and just added two extra dimensions: CUSTOMER_NATION and SUPPLIER_NATION. Another option would be to add the two columns—Customer's Nation and Supplier's Nation—in the fact table LINEITEM. But I won't or can't change the schema. However, in an ideal database world, my query would only have to read the light gray values.

Fooling Oracle

What would be really interesting would be to have two bitmap indexes in LINEITEM over fictitious columns, Customer's Nation and Supplier's Nation. That's when FBIs come into play. In Listing 2, I show how to implement the GET_CUST_NATION and GET_SUPP_NATION functions.

Listing 2. The DBMS_FBI package and functions GET_CUST_NATION and GET_SUPP_NATION.

```
CREATE OR REPLACE PACKAGE dbms_fbi IS
  cnt_cust NUMBER := 0;
  cnt_supp NUMBER := 0;
END dbms_fbi;
/

CREATE FUNCTION get_cust_nation (v_orderkey IN NUMBER)
                                  RETURN VARCHAR2
                                  DETERMINISTIC
                                  IS
  v_nation CHAR(25);
BEGIN
  dbms_fbi.cnt_cust := dbms_fbi.cnt_cust + 1;

  SELECT n.n_name INTO v_nation
    FROM orders o, customer c, nation n
   WHERE v_orderkey = o.o_orderkey
     AND o.o_custkey = c.c_custkey
     AND c.c_nationkey = n.n_nationkey;

  RETURN TRIM(v_nation); /* Because of white space */
END get_cust_nation;     /* in TPC-H data.         */
/

CREATE FUNCTION get_supp_nation (v_suppkey IN NUMBER)
                                 RETURN VARCHAR2
                                 DETERMINISTIC
                                 IS
  v_nation CHAR(25);
BEGIN
  dbms_fbi.cnt_supp := dbms_fbi.cnt_supp + 1;

  SELECT n.n_name INTO v_nation
    FROM supplier s, nation n
   WHERE v_suppkey = s.s_suppkey
     AND s.s_nationkey = n.n_nationkey;

  RETURN TRIM(v_nation); /* Because of white space */
END get_supp_nation;     /* in TPC-H data.         */
/
```

Listing 2 shows that although I've declared both

functions as deterministic, in fact they can't be considered as such—they read and write package variables (not necessary, but used to count the number of executions) and they read from the database. Oracle8i is less strict than previous versions regarding purity levels—it trusts the programmer more. In fact, there's even a new purity level called TRUST.

After creating the functions, the next step is to create the FBIs. Just before creating them, I initialize the counters:

```
BEGIN
  dbms_fbi.cnt_cust := 0;
  dbms_fbi.cnt_supp := 0;
END;
```

And now, I create the FBIs:

```
CREATE BITMAP INDEX idx_cust ON
  lineitem(SUBSTR(get_cust_nation(l_orderkey), 1, 25));

CREATE BITMAP INDEX idx_supp ON
  lineitem(SUBSTR(get_supp_nation(l_suppkey), 1, 25));
```

When using FBIs with functions returning strings, you have to SUBSTR the result. This happens because user-defined functions that return a string use VARCHAR2(4000) types. This is too big to be indexed, and Oracle will return "ORA-01450: maximum key length (758) exceeded." Functions returning numbers or dates don't need this.

The idea is to make the optimizer go and look for the result of GET_CUST_NATION and GET_SUPP_NATION in the FBIs, instead of executing the function. Let's see how many calls were made to each function during the indexes creation:

```
SET SERVEROUTPUT ON;

BEGIN
  dbms_output.put_line('CUST counter = ' ||
                       TO_CHAR(dbms_fbi.cnt_cust));
  dbms_output.put_line('SUPP counter = ' ||
                       TO_CHAR(dbms_fbi.cnt_supp));
END;

CUST counter = 6001215
SUPP counter = 6001215
```

Each function was executed once for every record in LINEITEM. Remember that there are 6,001,215 records in that table.

Now I'll make Oracle use the values from the FBIs in the original query. First, I must analyze both indexes:

```
ANALYZE INDEX idx_cust
  COMPUTE STATISTICS;

ANALYZE INDEX idx_supp
  COMPUTE STATISTICS;
```

Then I have to alter some session parameters in order for the FBIs to be used:

```
ALTER SESSION SET query_rewrite_enabled = true;
ALTER SESSION SET query_rewrite_integrity = trusted;
```

Comparing versions

The entire background job is done now except for one final step. I have to rewrite TPC-H query number 7. Listing 3 shows the new version of query 7.

Listing 3. Altered version of TPC-H query number 7.

```
SELECT
  sn supp_nation,
  cn cust_nation,
  l_year,
  sum(volume) revenue
FROM
  (
  SELECT /*+ INDEX(idx_cust)
             INDEX(idx_supp) */
    substr(get_supp_nation(l_suppkey), 1, 25) sn,
    substr(get_cust_nation(l_orderkey), 1, 25) cn,
    to_char(l_shipdate, 'YYYY') l_year,
    l_extendedprice * (1 - l_discount) volume
  FROM
    lineitem
  WHERE
    ((      substr(get_supp_nation(l_suppkey), 1, 25)
                                             = 'FRANCE'
       AND substr(get_cust_nation(l_orderkey), 1, 25)
                                             = 'GERMANY')
    OR (    substr(get_supp_nation(l_suppkey), 1, 25)
                                             = 'GERMANY'
       AND substr(get_cust_nation(l_orderkey), 1, 25)
                                             = 'FRANCE'))
     AND l_shipdate
         BETWEEN to_date('1995-01-01', 'yyyy-mm-dd')
             AND to_date('1996-12-31', 'yyyy-mm-dd')
 ) shipping
GROUP BY
  sn,
  cn,
  l_year
ORDER BY
  supp_nation,
  cust_nation,
  l_year;
```

In Listing 3, the differences from the original query are: the FROM clause refers just one (!) table now; n1.n_name was replaced by substr(get_supp_nation(l_suppkey), 1, 25); n2.n_name was replaced by substr(get_cust_nation(l_orderkey), 1, 25); and I added hints to the inner SELECT. There are also small differences in the alias I introduced just to make the code narrower. The differences are in italics.

The original query took 809 seconds (13 minutes and 29 seconds), and the new query took only 123 seconds (2 minutes and 3 seconds), representing a gain of more than 650 percent. This was the original execution plan:

```
Plan
-----------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
 SORT (GROUP BY)
  NESTED LOOPS
   NESTED LOOPS
    NESTED LOOPS
     NESTED LOOPS
      NESTED LOOPS
       TABLE ACCESS (FULL) OF 'ORDERS'
       TABLE ACCESS (BY INDEX ROWID) OF 'CUSTOMER'
```

```
      INDEX (UNIQUE SCAN) OF 'PK_CUSTOMER' (UNIQUE)
    TABLE ACCESS (BY INDEX ROWID) OF 'NATION'
     INDEX (UNIQUE SCAN) OF 'PK_NATION' (UNIQUE)
   TABLE ACCESS (BY INDEX ROWID) OF 'LINEITEM'
    INDEX (RANGE SCAN) OF 'PK_LINEITEM' (UNIQUE)
  TABLE ACCESS (BY INDEX ROWID) OF 'SUPPLIER'
   INDEX (UNIQUE SCAN) OF 'PK_SUPPLIER' (UNIQUE)
 TABLE ACCESS (BY INDEX ROWID) OF 'NATION'
  INDEX (UNIQUE SCAN) OF 'PK_NATION' (UNIQUE)
```

Here's the new execution plan:

```
Plan
----------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
 SORT (GROUP BY)
  TABLE ACCESS (BY INDEX ROWID) OF 'LINEITEM'
   BITMAP CONVERSION (TO ROWIDS)
    BITMAP OR
     BITMAP AND
      BITMAP INDEX (SINGLE VALUE) OF 'IDX_CUST'
      BITMAP INDEX (SINGLE VALUE) OF 'IDX_SUPP'
     BITMAP AND
      BITMAP INDEX (SINGLE VALUE) OF 'IDX_CUST'
      BITMAP INDEX (SINGLE VALUE) OF 'IDX_SUPP'
```

Am I fooling myself?

If you've reached this point, you might be asking, "But what if I update any table between ORDERS and NATION? What happens to the index?" The answer is nothing happens to the index. Therefore, the index won't be updated nor rebuilt upon changes in ORDERS, CUSTOMER, NATION, or SUPPLIER. It also means that an execution plan using the FBIs wouldn't yield the same results as an execution plan doing the nested loop over the five tables. That's why this technique is specially suited to data warehouses where updates in the dimensions are rare.

An interesting tidbit

Remember the counters from DBMS_FBI? After the execution of the new query (and upon re-initialization to zero), their values were:

```
CUST counter = 5924
SUPP counter = 5924
```

Thus, the functions were executed after all—but far fewer times than what would be expected. The subtlety is that query number 7 refers to Customer's Nation (and to Supplier's Nation) in both the WHERE and the SELECT clauses. What's interesting is that I can make Oracle use the indexes instead of executing the functions if the function call is in the WHERE clause. But if the function call appears in the SELECT clause, the function is really executed. Consequently, the executed plan isn't only going for the bitmap indexes. From the total of 6,001,215 records, only 5,924 (about 0.1 percent) will be selected using just the values stored in the indexes. For those 5,924 records of the inner SELECT, I'll call a function to calculate the Nations of Customer and Supplier. This happens because the system isn't intelligent enough. It knows the values of Customer's and Supplier's Nations for every record (stored in the FBIs), but it still calculates them again using the functions.

Food for thought

This particular query allows for an extra improvement. By re-writing it in some way, it's possible to remove the function calls from the inner SELECT clause. With this new improvement, the functions appear only in the WHERE clause, and that doesn't transform into a function call. Therefore, it's possible to have no function calls at all! Not even one. Can you figure it out?

For a hint, look at this new execution plan:

```
Plan
-----------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
 SORT (GROUP BY)
  VIEW
   SORT (UNIQUE)
    UNION-ALL
     TABLE ACCESS (BY INDEX ROWID) OF 'LINEITEM'
```

Continues on page 18

More on FBIs

- *Oracle8i SQL Reference:* http://technet.oracle.com/doc.pdf/server.815/a67779.pdf
- *Function Based Indexes,* by Thomas Kyte: http://govt.oracle.com/~tkyte/article1/index.html
- *Function Based Indexes,* by Chris Kempster: http://www.dbasupport.com/dsc/ora8/fbi.shtml
- *Oracle PL/SQL Language Pocket Reference,* by Steven Feuerstein, Bill Pribyl, and Chip Dawes, 1st Edition, April 1999

Avoid Costly Joins with FBIs . . .

Continued from page 6

```
     BITMAP CONVERSION (TO ROWIDS)
      BITMAP AND
       BITMAP INDEX (SINGLE VALUE) OF 'IDX_CUST'
       BITMAP INDEX (SINGLE VALUE) OF 'IDX_SUPP'
    TABLE ACCESS (BY INDEX ROWID) OF 'LINEITEM'
     BITMAP CONVERSION (TO ROWIDS)
      BITMAP AND
       BITMAP INDEX (SINGLE VALUE) OF 'IDX_CUST'
       BITMAP INDEX (SINGLE VALUE) OF 'IDX_SUPP'
```

This new query took only 114 seconds (1 minute and 54 seconds), and it's over 700 percent better than the original query.

The solution is available in this month's Source Code files at www.oracleprofessionalnewsletter.com.

Conclusions

If you're going to include this technique in your production environment, you must remember a few things:

- You have to create the functions and declare them as DETERMINISTIC.
- You have to create the indexes (bitmap or normal) using the functions.
- VARCHAR2 columns require special treatment using the SUBSTR function.
- The indexes have to be analyzed.
- You have to alter session parameters (QUERY_REWRITE_ENABLED, QUERY_REWRITE_INTEGRITY).
- You have to specify the indexes to use with optimizer hints.
- If you update data from any of the tables the indexes use, you have to drop and rebuild said tables, or the results might be different. ▲

DOWNLOAD BIZARRO.RTF at www.oracleprofessionalnewsletter.com

Pedro Bizarro is a graduate student at University Nova de Lisboa and a teaching assistant in all database courses at University of Coimbra, Portugal. He'll finish his master's by December 2000. He's already a Fulbright grantee for his Ph.D. program, starting September 2001. bizarro@dei.uc.pt.

Oracle Tip! Views with Sorts in Oracle8i

Jay Wortman

DOWNLOAD

ONE noteworthy new Oracle8i Release 8.1.5 feature is the ORDER BY clause in a view. Oracle almost hid this feature in the documentation. I remember my disbelief in my early days of writing Oracle code, when I discovered that the ORDER BY clause wasn't allowed in a view definition. I simply added the ORDER BY to the SELECT definition.

This table definition is useful for the examples:

```
CREATE TABLE Products(
   Prod        VARCHAR2(20),
   Code        NUMBER(5),
   Description VARCHAR2(40));
```

Here are view definitions that list the results in (ascending by default) Prod order and descending Code order:

```
CREATE VIEW Prod_Order_View AS
SELECT * FROM Products
ORDER BY Prod;

CREATE VIEW Code_Order_View_Desc AS
SELECT * FROM Products
ORDER BY Code DESC;
```

Selecting from the view is the same:

```
SELECT *
FROM Prod_Order_View;

SELECT *
FROM Code_Order_View_Desc;
```

Adding an ORDER BY when selecting from a view overrides the original definition. This SELECT lists the results in ascending order by CODE (though resulting in confusing code):

```
SELECT *
FROM Prod_Order_View
ORDER BY Code;
```

And here's a last example showing a (descending) Description column:

```
SELECT *
FROM Prod_Order_View
ORDER BY Description DESC;
```

The ORDER BY is valid in in-line views as well. ▲

DOWNLOAD WORTMAN.SQL at www.oracleprofessionalnewsletter.com

Jay Wortman is an independent consultant and has lived in Paris for more than 19 years. He ran his own consulting business in Paris for a number of years and now travels around Europe working on projects. He's an Oracle specialist (Oracle Certified DBA v7.3, v8.0 and v8.1), and his favorite projects include data warehouse and Web applications. He speaks fluent French and fluent Java. wortman_j@hotmail.com.