

Oracle

Solutions for High-End
Oracle® DBAs and Developers

Professional

Summer Reading, Data Caching, and String- Indexed Arrays

Steven Feuerstein

DOWNLOAD

In previous issues of *Oracle Professional*, Steven Feuerstein has written (with Bryn Llewellyn) about Oracle9i's new support for string-based indexes in associative arrays. Rather than indexing solely by `BINARY_INTEGER`, you can now index by hard-coded `VARCHAR2` types, `%TYPE` anchorings against database table columns and PL/SQL variables, and even `SUBTYPE`s based on `VARCHAR2` datatypes.

GIVEN the date (July 2003) and the temperature (I'm currently spending the week in Zurich and Bern, teaching best practices and new features to developers and DBAs at Swisscom; the Swiss are living through weeks of unrelenting heat), I thought it would be appropriate to follow up on earlier articles with an examination of how you can build and quickly scan your summer book reading list. In the process, I'll also explore how to apply best practices when using string-indexed collections as caches for database information. I'll finish up by taking a look at how you can use code generation techniques to avoid writing just about all of the code you'll need to take advantage of these features.

The "business" requirement

I love to read. My interests swing widely among science fiction/fantasy,

Continues on page 3

September 2003

Volume 10, Number 9

- 1 Summer Reading, Data Caching, and String-Indexed Arrays
Steven Feuerstein
- 7 Connecting Oracle to Microsoft Excel Using Oracle Objects for OLE
Tom Reid
- 10 Virtual Private Database
Sameer Wadhwa
- 16 Tip: Forcing Log Switches
Parin Jhaveri
- 16 September 2003 Downloads

DOWNLOAD

Indicates accompanying files are available online at
www.oracleprofessionalnewsletter.com.

**FULL PAGE AD:
QUEST**

Summer Reading...

Continued from page 1

mystery thrillers, and political analysis. I generally read more during the summer than the winter, since I've convinced myself that I should be lying on the beach, reading. Even if I don't get to the beach very often, I can still read! Being the PL/SQL-Oracle fanatic that I am, I've collected information about thousands of books that I might like to read in a relational table called, you guessed it, *book*. The definition of the book table is shown in Listing 1; I've defined a primary key, the ID column, as well as unique indexes for the ISBN number and the author-title combination. The book.tab file that's included in the accompanying Download (available at www.oracleprofessionalnewsletter.com) contains these DDL statements, as well as a script to populate the table with just under 10,000 rows.

Listing 1. Definition of the book table.

```
CREATE TABLE BOOK
(
  ID                INTEGER,
  ISBN              VARCHAR2(13 BYTE),
  TITLE            VARCHAR2(200 BYTE),
  SUMMARY          VARCHAR2(2000 BYTE),
  AUTHOR           VARCHAR2(200 BYTE),
  DATE_PUBLISHED  DATE,
  PAGE_COUNT      NUMBER
);

CREATE UNIQUE INDEX UN_BOOK_AUTHOR_TITLE ON BOOK
(TITLE, AUTHOR);

CREATE UNIQUE INDEX UN_BOOK_ISBN ON BOOK (ISBN);

ALTER TABLE BOOK ADD (
  PRIMARY KEY (ID));

ALTER TABLE BOOK ADD (
  CONSTRAINT UN_BOOK_AUTHOR_TITLE
  UNIQUE (TITLE, AUTHOR));

ALTER TABLE BOOK ADD (
  CONSTRAINT UN_BOOK_ISBN UNIQUE (ISBN));
```

Now, when I'm itching to get out to the beach and settle down for a good read, I don't want to have to wait a long time to generate my summer reading list. And while Oracle lets me query the database quite efficiently, I've also found that I can access my information so much *more* quickly by caching the data in a PL/SQL collection.

In previous years and versions of Oracle, I've utilized index-by tables that are indexed on BINARY_INTEGER to create a cached version of the book table. This approach allows me to very quickly retrieve a book if I have the primary key value. But I don't search for books by a sequence-generated key value. I search by author or title or, heck, even the ISBN number. (Don't laugh! When you're an author, you join the very elite crowd of human beings who understand the internal structure of the ISBN number and can find books of interest simply from

that information.)

Yet authors, titles, and ISBN "numbers" aren't numbers. They're strings (ISBNs have the form N-MMMMM-OOO-P). Thus, prior to Oracle9i Release 2, I had to write some fairly complicated code based on hash-based indexes (constructed with calls to DBMS_UTILITY.GET_HASH_VALUE) to quickly go from ISBN to book.

In the brave new world of Oracle, however, I can follow a much simpler, more intuitive path to get that same information in an instant. So let's explore how I can apply string-based indexes to get me out to the beach faster.

Interfaces come first

Before I start writing all the interesting associative array logic, I should first define the interface to this logic that I'll call to get the job done. It's extremely important to think through and stabilize the interface (names of programs, parameters, return values) *before* you start the implementation. If you dive too quickly into writing code, you'll undoubtedly waste an awful lot of time re-writing that code, as you experience such epiphanies as "Wait a minute; I forgot to pass in the X value! Darn, that changes everything!"

So what programs will I need to allow to flexibly and rapidly construct my reading list?

- Return a book for a primary key value (the ID column). This may not be the most user-friendly path to the data, but it will certainly come in handy.
- Return a book for a particular (unique) ISBN number.
- Return a book for a particular (unique) author-title combination.

Listing 2 contains the headers of programs for the summer reading package specification that fits these programs (see `summer_reading.pks` for the full definition). You pass in the appropriate values and return a row of book information. Well, that was easy enough. Note also that this package specification is designed to hide the method of implementation. I plan, in other words, to use cached associative arrays to retrieve the book records quickly, but you can't tell that this is the case by looking at the specification.

Listing 2. First pass at the book retrieval package specification.

```
CREATE OR REPLACE PACKAGE summer_reading
IS
  FUNCTION onebook (id_in IN book.id%TYPE)
  RETURN book%ROWTYPE;

  FUNCTION onebook (isbn_in IN book.isbn%TYPE)
  RETURN book%ROWTYPE;

  FUNCTION onebook (
    author_in book.author%TYPE
    , title_in book.title%TYPE)
  RETURN book%ROWTYPE;
```

This “information hiding” is a crucial aspect to high-quality software design. If I expose the implementation in any way, then any uses of this code will become dependent on that particular implementation. If I ever decide to *change* the implementation (perhaps Oracle12i will contain a new data structure that’s much, *much* faster than associative arrays), all uses of this code will also have to change.

Okay, the package specification is defined. Now I can move on to the implementation or package body.

Caching the book data

Let’s first look at the implementation of the primary key-based retrieval function. We don’t need any string-based associative arrays for this, but it forms, as you’ll soon see, the foundation for the other functions and associative arrays.

Note: All the elements of code you see in the rest of the article may be found, unless otherwise mentioned, in the `summer_reading.pkb` file.

At the very top of the package body, I declare an associative array type and instance of that type to store the cache of book information.

```
CREATE OR REPLACE
PACKAGE BODY summer_reading
IS
  TYPE id_aat IS
    TABLE OF book%ROWTYPE
    INDEX BY PLS_INTEGER;

  book_aa id_aat;
```

Once this collection is in place, I can load the contents of my relational table into the collection as follows:

```
PROCEDURE load_arrays
IS
BEGIN
  FOR rec IN (
    SELECT * FROM book )
  LOOP
    book_aa (rec.id) :=
      rec;
  END LOOP;
END load_arrays;
```

I then invoke this procedure in the initialization section of the package, thereby guaranteeing that the collection is populated, but only once in my session:

```
CREATE OR REPLACE
PACKAGE BODY summer_reading
IS
  ...
BEGIN
  load_arrays;
END summer_reading;
/
```

Before moving on to the implementation of the retrieval function, let’s take another look at the `load_arrays` procedure. Can you see any ways that you might be able to improve the performance of that procedure? As I copied the code into this article, I thought

to myself, “Wait a minute! A cursor FOR loop that populates a collection... isn’t that the perfect scenario for the BULK COLLECT feature?”

BULK COLLECT, which was first introduced in Oracle8i, allows you to retrieve multiple rows (or all of the rows) from a table with a single pass to the database or SQL layer. So I can, in a single line of code, ask for all the rows of `book` as follows:

```
SELECT *
  BULK COLLECT
  INTO book_aa
  FROM book;
```

When you’re talking about 10,000 rows of data, a switch to BULK COLLECT can yield an impressive gain in performance. But can I really use that here? I think not, at least not so simply as that. Here’s the problem: BULK COLLECT always populates a collection sequentially, using as the row numbers values 1 through N, where N is the number of rows retrieved. Yet I want to take advantage of the single INTEGER primary key as an “intelligent key” in my associative array. Thus, BULK COLLECT simply doesn’t fit my requirements—or, at least, it doesn’t directly and easily fit those requirements.

There *is* a way that I can still take advantage of BULK COLLECT in this situation: I can first transfer all of the rows from `book` to a collection. And then I can move those rows, deposited in sequentially allocated row numbers, into rows determined by the intelligence key value. Here are the changes I need to make in the package body to accomplish this:

1. Instantiate a *second* collection based on the `book_aat` type. Use this to hold the sequentially ordered data:

```
CREATE OR REPLACE
PACKAGE BODY summer_reading
IS
  TYPE id_aat IS
    TABLE OF book%ROWTYPE
    INDEX BY PLS_INTEGER;

  seq_book_aa id_aat;
  book_aa id_aat;
```

2. Modify the `load_arrays` procedure to first SELECT BULK COLLECT and then loop through the resulting contents, copying each row to the `book_aa` collection, specifying the primary key as the new row number.

```
PROCEDURE load_arrays
IS
BEGIN
  SELECT *
    BULK COLLECT
    INTO seq_book_aa
    FROM book;

  FOR indx IN
    seq_book_aa.FIRST ..
    seq_book_aa.LAST
  LOOP
    book_aa (
      seq_book_aa (indx).ID) :=
```

```

        seq_book_aa (indx);
    END LOOP;
END load_arrays;

```

Yes, this approach takes more code, but it works much more efficiently!

Retrieving one row for a primary key value

To finish off this first phase of implementation (focusing on the simplest aspect: primary key-based retrieval), here's the function that returns a single row:

```

FUNCTION onebook (
    id_in IN book.ID%TYPE
) RETURN book%ROWTYPE
IS
BEGIN
    RETURN book_aa (id_in);
END;

```

You certainly can't get much simpler than that, can you? Recall that the "old-fashioned" way of doing this same thing would look more like this:

```

FUNCTION onebook (
    id_in IN BOOK.ID%TYPE
)
RETURN BOOK%ROWTYPE
IS
    CURSOR onerow_cur
    IS
        SELECT*
        FROM BOOK
        WHERE
            ID = id_in
        ;
    onerow_rec BOOK%ROWTYPE;
BEGIN
    OPEN onerow_cur;
    FETCH onerow_cur
    INTO onerow_rec;
    CLOSE onerow_cur;
    RETURN onerow_rec;
END onebook;

```

Shrinking the volume doesn't always improve performance, but it *always* reduces the volume of code you have to maintain. In this case, you also get higher efficiency. That's enough to make a programmer *happy*!

Alternative paths to a book row

Great, so we have the basics in place: Pass in a primary key, get back a row. Now let's move on to the more interesting aspects of this task—emulating the unique indexes on the book table in string-based associative arrays. One unique index is ISBN, and the other is a concatenated index composed of two columns, author and title.

Working with string-based indexes, I can write code like this:

```

isbn_aa ('0-596-00180-0')
    .page_count := 750;

```

In other words, I simply pass the ISBN number as the row value for the associative array.

The situation with concatenated indexes is a bit more complex. Oracle doesn't support multi-dimensional

arrays. Thus, it's *not* possible to write code like this:

```

author_title_aa (
    'Feuerstein, Steven'
    , 'PLSQL: A Love Story')
    .page_count := 750;

```

I must supply a single string value. Can I therefore do the following (concatenate the two values together)?

```

author_title_aa (
    'Feuerstein, Steven'
    || 'PLSQL: A Love Story')
    .page_count := 750;

```

The answer is yes, I can do that, but I may not be very happy with the results. Suppose, for example, that two authors are named "SAM" and "SAMMY". SAM writes a book called "MYLIFE". SAMMY writes a book called "LIFE". As you can readily see, when I concatenate these two unique combinations together, they merge into the same string "SAMMYLIFE".

Thus, when working with indexes based on multiple columns, it's extremely important to concatenate a delimiter (and always the *same* delimiter) between the two values. Furthermore, this delimiter should never appear in any of the column values. That way, I guarantee that all combinations are unique.

Recognizing this need, I'm going to create in my package body two special elements: a SUBTYPE and a function. The SUBTYPE defines, in essence, a new application-specific datatype for use in my package:

```

SUBTYPE author_title_t
    IS VARCHAR2 (32767);

```

It's really nothing more than an alias for a VARCHAR2 declaration. I'd love to be able to define the "author, title" combination based on an existing column using %TYPE, but it's a derived value. So I use the SUBTYPE to give a name and type to this derived value. You'll soon see how it will come in very handy in my package body.

I then define a function that returns an author-title combination constructed in a consistent fashion, from the individual elements:

```

FUNCTION author_title (
    author_in book.author%TYPE
    , title_in book.title%TYPE
    , delim_in IN VARCHAR2 := '^'
)
RETURN author_title_t
IS
BEGIN
    RETURN
        UPPER (author_in)
        || delim_in
        || UPPER (title_in);
END;

```

Notice the return type of the function: `author_title_t`. I could have simply returned VARCHAR2, but I use the SUBTYPE to self-document very clearly and precisely

what the function is sending back. Now, by using the function and never concatenating values myself explicitly, I'm guaranteed a consistent and unique row value for my string-based index.

Now it's time to declare those string-based indexes, based of course on new TYPE declarations. Here they are, along with the previous declarations in the package body:

```

1 CREATE OR REPLACE
2 PACKAGE BODY summer_reading
3 IS
4     SUBTYPE author_title_t
5     IS VARCHAR2 (32767);
6
7     TYPE id_aat IS
8     TABLE OF book%ROWTYPE
9     INDEX BY PLS_INTEGER;
10
11     TYPE isbn_aat IS
12     TABLE OF book.ID%TYPE
13     INDEX BY book.isbn%TYPE;
14
15     TYPE author_title_aat IS
16     TABLE OF book.ID%TYPE
17     INDEX BY author_title_t;
18
19     seq_book_aa id_aat;
20     book_aa id_aat;
21     by_isbn_aa isbn_aat;
22     by_author_title_aa author_title_aat;

```

Let's take a closer look at these new TYPE declarations. The first, on lines 11-13, creates a string-based index type, using the ISBN value as the index type. Each row of that collection stores only the primary key value, not the whole row. I *could* have defined this TYPE as a collection of records, but that would use more memory than is necessary. The book_aa array already contains all of the book information. The array indexed by ISBN values needs only a pointer *to* the appropriate row in book_aa (this will be arranged in the enhanced load_arrays procedure).

The second TYPE declaration, on lines 15-17, also contains only book primary key values. But notice the datatype used for the index: author_title_t. Again, I could simply use something like INDEX BY VARCHAR2(1000), but by reusing my SUBTYPE, I self-document the type of value that's going to be used as the index.

Okay, so I now have four different collections defined. Let's build the functions that return a book for either the ISBN number or author-title combination. First, the ISBN-based retrieval:

```

FUNCTION onebook (
    isbn_in IN book.isbn%TYPE)
RETURN book%ROWTYPE
IS
    l_id book.ID%TYPE;
BEGIN
    l_id :=
        by_isbn_aa (isbn_in);
    RETURN onebook (l_id);
END;

```

Another short and sweet program! First, I use the ISBN number to get the ID for that book from the string-based index. Next, I use that book ID as the

input to the original onebook function to retrieve all the information for that ID. Okay, now the author-title-based retrieval:

```

FUNCTION onebook (
    author_in book.author%TYPE
    , title_in book.title%TYPE
)
RETURN book%ROWTYPE
IS
    l_id book.ID%TYPE;
BEGIN
    l_id :=
        by_author_title_aa (
            author_title (
                author_in
                , title_in));
    RETURN onebook (l_id);
END;

```

We have a bit more work to do in this version. The user passes in the author and title. I then call the function, author_title, which encapsulates the "rule" for constructing the string row value. I then use that concatenated value to retrieve the book ID from the by_author_title_aa collections. Finally, I again call the original function to pass back the whole record.

The gain for this pain

I've talked about dramatic gains in improvement when using collection-based caching. I should back that up with some tests. You'll find in the summer_reading package a procedure named TEST. You can use this to compare the performance of fetch functions that always query from the database (and, therefore, retrieve the information from the SGA, or System Global Area) vs. onebook routines that return the data from associative arrays (and, thus, from the PGA, or Program Global Area). **Table 1** shows the results for a test run of 100,000 iterations. As one would expect, query times for the primary key are fastest (regardless of method). The author-title retrieval is the slowest, with a much wider differential exhibited in the associative array implementations.

Table 1. Comparing performance of database and associative array fetches.

Number of iterations: 100,000

Fetch type	From SGA	From PGA
By primary key	16.89	2.69
By ISBN	16.97	4.47
By author-title	17.1	7.13

Let's get practical

You might easily agree that the performance gains shown in the previous section are worthy. That doesn't mean you're going to be building packages like summer_reading anytime soon. It is, quite frankly, probably just too much of a bother. Recognizing that fact, I decided to make things much easier; to make writing this code so

Continues on page 15

Connecting Oracle to Microsoft Excel Using Oracle Objects for OLE

Tom Reid

It's often incredibly useful to be able to process the data held in an Oracle database with the features found on PC spreadsheets like Excel. Graphing in particular comes to mind—developers who have tried using the Oracle Graphics Builder product will know what I mean. Or perhaps you have clients that need to receive data in a specific format such as CSV. On the other hand, maybe clients send you data in Excel format that you need to store in your Oracle database. Whatever the reasons, if you need to get your Oracle data into Excel or vice versa, you have a number of programmatic options available, the most common of which is ODBC. However, a much less well-known method but one that's simpler to code uses Oracle Objects For OLE, or OO4O. Tom Reid shows the way.

ORACLE Objects For OLE (OO4O) is a piece of middleware developed by Oracle to allow native connectivity from Oracle databases to Windows COM Automation technology. What this means in practice is that, for a whole host of PC applications such as Visual C++, VBA (Excel), FoxPro, and others, a set of APIs becomes available to enable us to open Oracle database connections and run arbitrary DDL and DML statements against the database from within the Windows client. Getting OO4O is easy, as a copy should have been shipped along with your Oracle distribution. However, if you can't find it, go to Oracle's Technology Network Web site (<http://otn.oracle.com/software/tech/windows/ole/content.html>), where you can download the version appropriate to your installation for free. You should note that OO4O only works with Excel version 5 or later.

Performance considerations

I'm often asked how fast OO4O is in comparison to other methods, such as ODBC and OLE DB, that can be used to get data to and from Excel and Oracle. I freely admit that I don't have a definitive answer to that question, as OO4O is the only method I've ever used. However, one organization that should know the answer is Oracle itself, and the following passage is taken directly

from its Technology Network Web site:

Because it is a native driver, OO4O generally provides the fastest performance on Windows clients to Oracle databases. It does not incur the overhead of ODBC and OLE DB drivers. OO4O has been developed and has evolved specifically for use with Oracle database servers. It provides easy access to features that are unique to Oracle, but are otherwise cumbersome or inaccessible to use from ODBC and OLE DB-based components, such as ADO.

If it's good enough for Oracle, it's good enough for me.

Using OO4O

As usual with these things, the easiest way to gain an understanding is by looking at some sample code. I'm going to present four short snippets of VBA code contained in macros that will highlight the most common types of Oracle-to-Excel interaction you're likely to use. They'll also be a useful starting point should you need to code more complex applications later on. Note that for brevity I haven't included any error-checking code here, as error checking is dealt with in a later section of this article. The code in the listings that follow have been tested on the following systems:

- NT4, Excel 97, OO4O version 3.13 and Oracle 8.1.6
- NT4, Excel 97, OO4O version 2.0 and Oracle 7.3.3

The first example ([Listing 1](#)) shows the use of a Data Definition Language (DDL) statement to drop a table in the Oracle database. Be aware that if you don't have permission to drop a table in the database, this statement will fail just as readily in OO4O as it would using SQL*Plus or any other tool connected to the database. To begin with, start up Excel and select the Tools | Macro | Macros menu option. Type in an appropriate name for the macro, such as *droptab*, and hit the Create button. Enter the code contained in Listing 1.

Listing 1. Use OO4O to issue a drop table statement.

```
' Use OO4O to run a drop table statement

Sub droptab()

    Dim objsession as object
    Dim objdatabase as object

    Set objsession = _
        CreateObject("oracleinprocserver.xorasession")
    Set objdatabase = _
        objsession.opendatabase("mydb","myuser/mypass",0&)

    ddlstr = "drop table mytab"
    objdatabase.executeSQL (ddlstr)

    ' Tidy up
    '
    set objsession = nothing
    set objdatabase = nothing

End Sub
```

The first two *set* statements are common to almost all OO4O code and simply make a connection to the database and return a handle to it. The *opendatabase* statement takes three parameters. The first two are the familiar database name and connect string. The third parameter is used to set optional modes for opening the database. In this case, we tell OO4O to open the database in its default mode. The second two lines of code should be fairly self-explanatory—set up the database command and then execute it. Now, by simply running this macro from within Excel, the named table in your Oracle database will be dropped.

My second example shows how to get data contained in your spreadsheet into an Oracle database. For instance, suppose your head office has sent you an Excel spreadsheet that contains new monthly sales figures for all salespeople in your organization. Typically, such a spreadsheet might contain a list of salesperson IDs in column A and their corresponding sales figures in column B. Your job is to update the sales table in your Oracle database with the new sales figures. **Listing 2** displays how we can accomplish this using OO4O.

Listing 2. Getting data from Excel into Oracle.

```
Sub updatetab()

    Dim objsession as object
    Dim objdatabase as object

    Set objsession = _
        CreateObject("oracleinprocserver.xorasession")
    Set objdatabase = _
        objsession.opendatabase("mydb","myuser/mypass",0&)

    t = 1

    ' loop until no more salesperson IDs available

    While (Cells(t, 1) <> "")

        dmlstr = "update sales set sale_value = " & _
            Cells(t, 2) & " where sales_id = " & _
            Cells(t, 1)
```

```
        objdatabase.executeSQL (dmlstr)

        t = t + 1

    Wend

    ' Tidy up
    '
    set objsession = nothing
    set objdatabase = nothing

End Sub
```

There's a bit more VBA code in this example, but it should still be fairly easy to see what's going on here. The first two lines, as before, create a database connection and return a handle to it. Next, we have a simple loop. Starting at cell A1, you construct an update statement based on the values contained in cells A1 and B1. Execute the resulting update statement and move down to the next row in the spreadsheet. Repeat until there are no more salesperson IDs to process.

My third example is probably the one you'll find most useful, as it shows how to get data contained in your Oracle database into the rows and columns of your Excel spreadsheet. There's slightly more complexity required, because we need to deal with returned data. So, continuing with the second example, suppose you now have to send back to the head office a spreadsheet containing a list of the total sales figures for the current year broken down by month. **Listing 3** shows how you can achieve this.

Listing 3. Getting data from Oracle into Excel.

```
Sub selectdata()

    Dim objsession as object
    Dim objdatabase as object

    Set objsession = _
        CreateObject("oracleinprocserver.xorasession")
    Set objdatabase = _
        objsession.opendatabase("mydb","myuser/mypass",0&)

    selstr = "select SUM(sales_val) Sales,"
    selstr = selstr & "month from sales_curr_year "
    selstr = selstr & "group by month"

    Set oradynaset = _
        objdatabase.dbcreatedynaset(selstr, 0&)

    if oradynaset.RecordCount > 0 then

        ' We have records to process so
        ' move to the first record in the set
        '
        oradynaset.movefirst

        ' Omit this FOR loop if you don't want
        ' to print out column headings
        '
        For x = 1 To oradynaset.fields.Count

            Cells(1, x) = _
                oradynaset.fields(x - 1).Name

        Next x

        ' for each record
        '
```

```

For i = 1 To oradynaset.RecordCount
    ' for each field
    '
    For x = 1 To oradynaset.fields.Count
        Cells(i + 1, x) = _
            oradynaset.fields(x - 1).Value
    Next x

    oradynaset.movenext
Next i

End If

' Tidy up
'
set objsession = nothing
set objdatabase = nothing

End Sub

```

The main difference in this example is the use of the *oradynaset* object, which OO4O uses to collect data from the database. This takes two parameters. The first is the select statement we're sending to Oracle, and the second, like in the *opendatabase* statement, retrieves the data set in default mode. The rest of the code is fairly straightforward. The first *For* loop prints out a row of field titles in the first row of the spreadsheet. Next, we loop through each record and assign the first field value into cell A2, the next into cell B2, and so on. When all of the fields for the current record have been processed, you go to the next record in the set and start assigning values into the cells of the next row of the spreadsheet. This is repeated until there aren't any records left to be processed.

My final example is slightly more involved, but it shows just how versatile OO4O can be. Here we'll use it to both create and call a stored database procedure. Our stored procedure, called *updsal*, takes two arguments: a number IN and a varchar2 OUT. Our stored procedure will simply update a column in a table using the value in our first parameter and return a string containing the number of records that were updated in our second parameter (see [Listing 4](#)).

Listing 4. Using OO4O to create and call a stored procedure.

```

' Parameter description values
Const ORA_INPUT = 1
Const ORA_OUTPUT = 2
Const ORA_BOTH = 3
Const ORA_VARCHAR2 = 1
Const ORA_NUMBER = 2

Sub call_a_proc()
    Dim objsession As Object
    Dim objdatabase As Object

    Set objsession = _
        CreateObject("oracleinprocserver.xorasession")
    Set objdatabase = _
        objsession.opendatabase("mydb", "myuser/mypass", 0&)

    ' create our procedure
    '
    sqlstr = "create or replace procedure updsal "
    sqlstr = sqlstr & "(pct_raise in number,"

```

```

sqlstr = sqlstr & "nr_updated out varchar2) as "
sqlstr = sqlstr & "begin update salary set emp_sal ="
sqlstr = sqlstr & "emp_sal * (100+pct_raise)/100;"
sqlstr = sqlstr & "nr_updated:='Records updated = "
sqlstr = sqlstr & "'|| sql%rowcount; end;"

retval = objdatabase.ExecutesQL(sqlstr)

' Now we have to set up our parameters, specify
' types, modes, values, etc...
' I'm feeling generous - give everyone a 50% pay raise
'
objdatabase.Parameters.Add "pct_raise", "", _
    ORA_INPUT
objdatabase.Parameters("pct_raise").ServerType = _
    ORA_NUMBER
objdatabase.Parameters("pct_raise").Value = 50

objdatabase.Parameters.Add "nr_updated", "", _
    ORA_OUTPUT
objdatabase.Parameters("nr_updated").ServerType = _
    ORA_VARCHAR2

' Run updsal using an anonymous PL/SQL block
'
sqlstr = "begin updsal(:pct_raise,:nr_updated); end;"
ret_val = objdatabase.ExecutesQL(sqlstr)

' print out our returned variable, it should display
' something like:- Records Updated = 10
'
MsgBox objdatabase.Parameters("nr_updated").Value

' Tidy things up
'
objdatabase.Parameters.Remove "pct_raise"
objdatabase.Parameters.Remove "nr_updated"

Set objdatabase = Nothing
Set objsession = Nothing

End Sub

```

As before, we set up our database connection and then constructed and created a stored database procedure. It's not clear from the listing what the procedure looks like, so I've reproduced it here in a more readable format:

```

create or replace procedure updsal(pct_raise IN number
,nr_updated OUT varchar2) as
begin
    update salary set emp_sal = emp_sal *
        (100+pct_raise)/100;
    nr_updated:='Records updated = '|| sql%rowcount;
end;

```

Next, we have to tell Oracle what our procedure parameters are. We define their type (number or varchar2), their mode (IN or OUT), and, in the case of our first parameter, we also set its value to pass to the procedure. After this we can simply make a call to our stored procedure using an anonymous PL/SQL block. Finally, if there are no exceptions, we display the number of records we've just updated.

OO4O error handling

When dealing with OO4O, the two main approaches to error handling are:

- Just treat OO4O errors the same way as you would

Continues on page 14

Virtual Private Database

Sameer Wadhwa

In this article, Sameer Wadhwa discusses how Virtual Private Database (VPD) can enhance database security. He also walks through the steps of configuring and testing VPD.

WHEN you talk about a database, one of the most important topics that comes to mind is database security. Can hackers use any third-party tool to see and break into my database? Is my data secured outside my application? How do I increase data security?

Virtual Private Database (VPD) is the answer to all of these questions.

VPD was introduced in Oracle8i. It's become more popular and useful with the additional features in Oracle9i. In essence, it segregates data by providing a mechanism to view or manipulate information that's relevant to an individual user only. There's no need to implement security in each application that accesses data because the security rules are applied to the database server only, and that's why VPD is completely safe and secure. So it's a good option if your business needs require data access to the outside world or an unsecured environment, where you want to make your own security rules to give access to your customer or end user according to your business requirement, without worrying about leaking any type of confidential information. This feature is most commonly used in the industries of banking, finance, Web hosting, manufacturing, pharmaceuticals, and defense.

Let's consider an example where you want to implement VPD or row-level security, as illustrated in Figure 1. Database users JAMES and BLAKE each issue a select statement against the EMP table owned by the SCOTT user, but can only view data that's relevant to them. From the user's point of view, the set of rows that he or she has access to only exists inside the table. Each user should only be allowed to do modifications on the rows that follow the security rules. Based upon these security rules, Oracle generates a predicate clause that transparently appends to the user's SQL statement. This concept is called Virtual Private Database.

In the preceding example, when JAMES issues a query such as this:

```
select * from SCOTT.EMP
```

Oracle automatically translates this to the following SQL statement:

```
SELECT "EMPNO","ENAME","JOB","MGR","HIREDATE",
" SAL","COMM","DEPTNO"
FROM "SCOTT"."EMP" "EMP"
WHERE (EMPNO=SYS_CONTEXT('SCOTTCTX','EMP_ID'))
```

Here, (EMPNO=SYS_CONTEXT('SCOTTCTX','EMP_ID')) is the predicate clause added by Oracle based upon the defined security policy. You'll learn later in this article how to add a security policy with the database table.

Architecture of VPD

Figure 2 (on page 11) illustrates the architecture of VPD. Whenever a user connects to the database, the database logon trigger fires, which calls the CONTEXT stored procedure to set the defined context. Policy rules are defined according to the user CONTEXT. After connection, when the user issues a query or DML statement against a policy-linked table, Oracle determines which policy needs to be linked according to the statement type. In other words, you can link various policies depending on the select, insert, update, or delete statement. The policy further indicates which function to call to implement security rules. If you have different policy rules for selecting, updating, inserting, or deleting data, you can create different functions to form a predicate. Depending upon the policy rules defined in the predicate procedure, the resultant predicate is appended to the user's query. The server process now executes this predicated query and sends

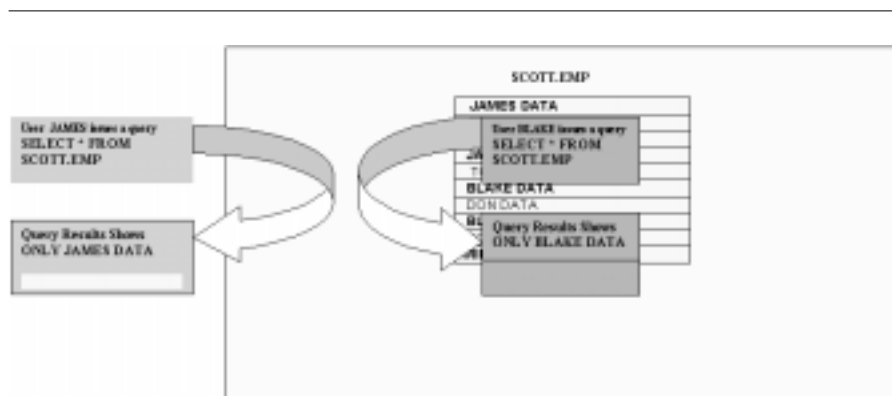


Figure 1. VPD applied a rule where users can see only their data.

the result back to the user.

Implementing VPD

Here are the major steps to implement row-level security thorough VPD:

1. (Option step) Configure the database accounts for demo purposes.
2. Assign the Create Any Context privilege to the owner of the object on which you want to provide row-level security.
3. Create a context package and grant to public.
4. Create a database logon trigger to execute the context package.
5. Verify the results from SESSION_CONTEXT.
6. Create security policies and functions.
7. Add policies to the table.

Let's run through these steps and implement our security policy.

Step 1: Configure database accounts

First, let's create two database users for demonstration. Let's also create a control table in the SCOTT schema. Note that the control table is called EMPIDCONTROLTAB and it contains the EMPNO and ENAME columns from the SCOTT scheme's EMP table.

```
-- Connect as SYS database user

SQL> CREATE USER JAMES IDENTIFIED BY JAMES123;
SQL> CREATE USER BLAKE IDENTIFIED BY BLAKE123;
SQL> GRANT CREATE SESSION TO JAMES, BLAKE;
SQL> CREATE TABLE SCOTT.EMPIDCONTROLTAB AS
      SELECT EMPNO, ENAME FROM SCOTT.EMP;
SQL> GRANT SELECT ON SCOTT.EMP TO JAMES, BLAKE;
```

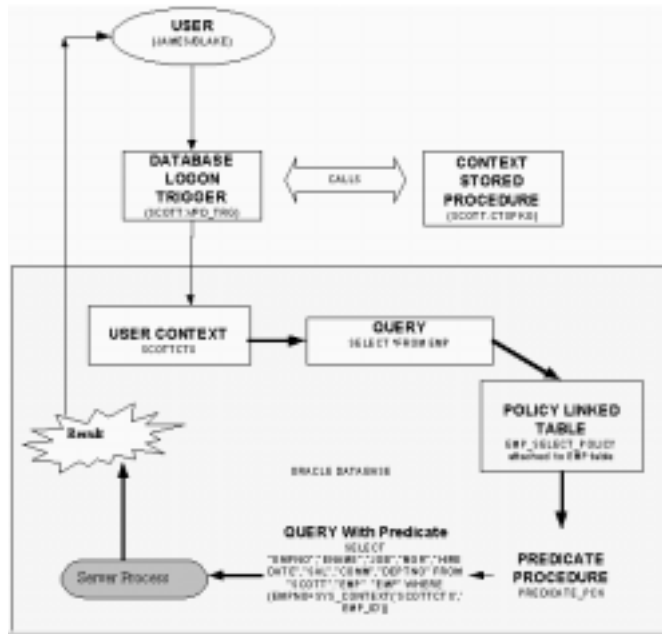


Figure 2. Architecture of VPD.

Step 2: Assign system privilege

Assign the Create Any Context privilege to the owner of the object on which you want to provide row-level security:

```
-- Connect as SYS database user

SQL> GRANT CREATE ANY CONTEXT TO SCOTT;

Grant succeeded.
```

Step 3: Create a context package and grant to public

Next we create a context package and grant to public:

```
-- Connect as SCOTT database user

SQL> CREATE OR REPLACE CONTEXT SCOTTCTX USING SCOTT.CTXPKG;

Context created.

SQL> CREATE OR REPLACE PACKAGE SCOTT.CTXPKG AS PROCEDURE
      CTXPROC;
END;
/

Package created.
```

In the following package, I'm setting the attributes of the context. The rules are as follows. If the session user is SCOTT, set the attribute OWNER to TRUE in SCOTTCTX. If the session user isn't SCOTT, find its ID in the control table and set the attribute EMPID with the EMPNO of the session user. If we can't find the ID in the control table, set the attribute EMPID to 0.

```
SQL> CREATE OR REPLACE PACKAGE BODY SCOTT.CTXPKG
IS
  PROCEDURE CTXPROC
  IS
    VEMPNO    NUMBER;
  BEGIN
    -- set the attribute OWNER=TRUE if user is SCOTT
    IF SYS_CONTEXT ('USERENV',
                   'SESSION_USER') = 'SCOTT'
    THEN
      DBMS_SESSION.SET_CONTEXT ('SCOTTCTX',
                                'OWNER', 'TRUE');
    ELSE
      BEGIN
        SELECT EMPNO
        INTO VEMPNO
        FROM SCOTT.EMPIDCONTROLTAB
        WHERE ENAME = SYS_CONTEXT ('USERENV',
                                  'SESSION_USER');
        -- set the attribute EMP_ID with the EMPNO of the
        -- session user
        DBMS_SESSION.SET_CONTEXT ('SCOTTCTX',
                                  'EMP_ID', VEMPNO);
      EXCEPTION
        WHEN NO_DATA_FOUND
        THEN
          /* NO EMPLOYEE ID FOUND, SET A DUMMY ID
          */
          DBMS_SESSION.SET_CONTEXT ('SCOTTCTX',
                                    'EMP_ID', 0);
      END;
    END IF;
  END CTXPROC;
END CTXPKG;
/
```

Package body created.

```
SQL> GRANT EXECUTE ON SCOTT.CTXPKG TO PUBLIC;
```

Grant succeeded.

Step 4: Create a database logon trigger

Now we'll create a database logon trigger to execute the context package:

```
-- Connect as SYS database user

CREATE OR REPLACE TRIGGER SCOTT.VPD_TRG
AFTER LOGON ON DATABASE
BEGIN
    SCOTT.CTXPKG.CTXPROC;
END;
/
Trigger created.
```

Step 5: Verify the results

It's time to verify the results from SESSION_CONTEXT:

```
SQL> connect james/james123
Connected.
SQL> col attribute format a10
SQL> col value format a10
SQL> col attribute format a10
SQL> select * from SESSION_CONTEXT;
```

NAMESPACE	ATTRIBUTE	VALUE
SCOTTCTX	SETUP	FALSE
SCOTTCTX	EMP_ID	7900

```
SQL> connect blake/blake123
Connected.
SQL> select * from SESSION_CONTEXT;
```

NAMESPACE	ATTRIBUTE	VALUE
SCOTTCTX	SETUP	FALSE
SCOTTCTX	EMP_ID	7698

```
SQL> connect scott/tiger
Connected.
SQL> select * from SESSION_CONTEXT;
```

NAMESPACE	ATTRIBUTE	VALUE
SCOTTCTX	OWNER	TRUE

You can see the SESSION_CONTEXT view will show you the appropriate attribute value corresponding to the user session.

Step 6: Creating security policies and functions

This is the most important step, where you'll assign or generate a predicate. This generated predicate will append to the WHERE clause of the SELECT statements. Once defined, this will happen automatically. The appended SQL query will now retrieve the relevant information from the database.

The example package in Listing 1 will generate the predicate based upon the user context defined by the SCOTTCTX context. The following rules are implemented by the predicate procedure:

- Rule 1: An unauthorized

user shouldn't be able to see the result.

- Rule 2: A user can see only his or her own records.
- Rule 3: The OWNER of the table should see all of the records.

For Rule 1, if the user isn't authorized—that is, the SYS_CONTEXT ('SCOTTCTX', 'EMP_ID') function returns 0—the predicate will be set to '1=2', which will result in a FALSE condition; hence, the unauthorized user won't see the result. For Rule 2, The SYS_CONTEXT ("SCOTTCTX", "EMP_ID") will return the correct value of the attribute EMP_ID, which will match the EMPNO and allow the query to execute. For Rule 3, we set the PREDICATE to NULL, which will allow the user to query without any restriction.

Step 7: Add policy to table

Once the policy is ready, you have to attach it to the table with the help of the DBMS_RLS.ADD_POLICY procedure.

```
-- Connect as SYS database user

BEGIN
    DBMS_RLS.ADD_POLICY
        (OBJECT_SCHEMA      => 'SCOTT',
         OBJECT_NAME        => 'EMP',
         POLICY_NAME        => 'EMP_SELECT_POLICY',
         FUNCTION_SCHEMA    => 'SCOTT',
         POLICY_FUNCTION    => 'PREDICATE_PCK.EMP_SELECT',
         STATEMENT_TYPES   => 'SELECT',
         ENABLE              => TRUE
        );
END;
/
```

Listing 1. Generating the predicate.

```
-- Connect as SCOTT database user

CREATE OR REPLACE PACKAGE PREDICATE_PCK
AS
    FUNCTION EMP_SELECT (OWNER VARCHAR2, OBJNAME VARCHAR2)
        RETURN VARCHAR2;
END PREDICATE_PCK;
/

CREATE OR REPLACE PACKAGE BODY PREDICATE_PCK
IS
    FUNCTION EMP_SELECT (OWNER VARCHAR2, OBJNAME VARCHAR2)
        RETURN VARCHAR2
    IS
        PREDICATE VARCHAR2 (2000);
    BEGIN
        IF (SYS_CONTEXT ('SCOTTCTX', 'EMP_ID') = 0)
            THEN
                PREDICATE := '1=2';
            ELSE
                PREDICATE := 'EMPNO=SYS_CONTEXT(''SCOTTCTX'', ''EMP_ID'')';
            END IF;

        IF (SYS_CONTEXT ('SCOTTCTX', 'OWNER') = 'TRUE')
            THEN
                PREDICATE := NULL;
            END IF;

        RETURN PREDICATE;
    END EMP_SELECT;
END PREDICATE_PCK;
/

SQL> grant execute on SCOTT.PREDICATE_PCK to public;

Grant succeeded.
```

For the INSERT, UPDATE, or DELETE SQL statement, you have to add the UPDATE_CHECK parameter in DBMS_RLS.ADD_POLICY.

```
BEGIN
  DBMS_RLS.ADD_POLICY
    (OBJECT_SCHEMA      => 'SCOTT',
     OBJECT_NAME        => 'EMP',
     POLICY_NAME        => 'EMP_UPDATE_POLICY',
     FUNCTION_SCHEMA    => 'SCOTT',
     POLICY_FUNCTION    => 'PREDICATE_PCK.EMP_UPDATE',
     STATEMENT_TYPES   => 'UPDATE,INSERT',
     UPDATE_CHECK       => 'TRUE',
     ENABLE             => TRUE
    );
END;
/
```

Verifying the VPD configuration

VPD is now implemented; let's take a look at the result. According to our implemented VPD rules, other than SCOTT, each user should see only his or her rows.

First, connect as JAMES (see Listing 2). Next, connect as BLAKE (see Listing 3). Then let's see what happens if we connect as SCOTT—the owner of the table (see Listing 4).

Finally, let's connect as HR, who has no security policy set up. Also assume that the HR database user already exists.

```
SQL> CONNECT SYS/***** AS SYSDBA
SQL> GRANT SELECT ON SCOTT.EMP TO HR;
SQL> CONNECT HR/**
SQL> select * from SCOTT.EMP;
no rows selected
```

Getting more information

The VSSQL and VSSQLAREA dynamic views don't show you the predicated queries. Even session sql*trace and event 10046 don't give you much information. To know the exact query with predicate, you have to set the event 10730 at level 12. Also make sure you run the DBMS_RLS.REFRESH_POLICY procedure, as event 10720 won't create the trace file if SQL is already parsed in the shared pool.

```
SQL> execute DBMS_RLS.REFRESH_POLICY;
SQL> connect james/james123
Connected.
SQL> ALTER SESSION SET EVENTS
      '10730 trace name context forever, level 12';
Session altered.
SQL> select * from scott.emp;
```

The trace file in Listing 5 will be generated in the

Listing 2. Test Case 1—connect as JAMES.

```
SQL> CONNECT JAMES/JAMES123
Connected.
SQL> select * from SCOTT.EMP;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7900	JAMES	CLERK	7698	03-DEC-81	950		30

Listing 3. Test Case 2—connect as BLAKE.

```
SQL> CONNECT BLAKE/BLAKE123
Connected.
SQL> select * from SCOTT.EMP;
SQL> select * from SCOTT.EMP;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30

Listing 4. Test Case 3—connect as SCOTT.

```
SQL> CONNECT SCOTT/TIGER
Connected.
SQL> select * from SCOTT.EMP;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

udump destination.

The data dictionary view in Listing 6, as well as

Oracle Enterprise Manager (OEM), will give you more information.

Listing 5. The trace file.

```
Windows thread id: 280, image: ORACLE.EXE

*** SESSION ID:(8.970) 2003-04-11 15:50:45.000
-----
Logon user      : JAMES
Table/View     : SCOTT.EMP
Policy name    : EMP_SELECT_POLICY
Policy function: SCOTT.PREDICATE_PCK.EMP_SELECT
RLS view      :
SELECT "EMPNO","ENAME","JOB","MGR","HIREDATE","SAL","COMM","DEPTNO" FROM
"SCOTT"."EMP" "EMP" WHERE (EMPNO=SYS_CONTEXT('SCOTTCTX','EMP_ID'))
```

Listing 6. The data dictionary view.

```
DBA_CONTEXT
DBA_POLICIES

SQL> select * from dba_context;

NAMESPACE  SCHEMA          PACKAGE          TYPE
-----
LT_CTX     SYS             LT_CTX_PKG      ACCESSED LOCALLY
SCOTTCTX   SCOTT          CTPXPKG         ACCESSED LOCALLY

SQL> select policy_name,function,sel,ins,upd,del from dba_policies;

POLICY_NAME  FUNCTION        SEL      INS      UPD      DEL
-----
EMP_SELECT_POLICY  EMP_SELECT      YES      NO      NO      NO
```

Conclusion

Virtual Private Database is a very secure solution to provide database security at the row level. It has a lower cost of ownership and removes application security problems. ▲

Sameer Wadhwa is an Oracle Certified 9i DBA. He has master's degree in mathematics and has more than nine years of experience in Oracle Database Technology. His work includes advanced performance tuning, capacity planning, database security, and physical/logical database design in Unix and Windows environments. He's currently working with NuGenesis Technologies Corporation, in Westborough, MA. He's also Webmaster of his technical Web site, "Oracle Techniques" (www.SamOraTech.com). SamOracle@Yahoo.com.

Connecting Oracle...

Continued from page 9

regular VBA errors by constructing a generic VBA error handler. In this case, the subroutine from Listing 2 would look something like that shown in Listing 5.

- If you want to specifically catch and deal with OO40 type errors, you can use the *LastServerError* and *LastServerErrortext* properties of the database and session objects. See Listing 6 for an example.

Listing 5. Using generic VBA error handling with OO40.

```
Sub selectdata()

Dim objsession as object
Dim objdatabase as object

on error goto err_handler

Set objsession = _
CreateObject("oracleinprocserver.xorasession")
Set objdatabase = _
objsession.opendatabase("mydb","myuser/mypass",0&)

selstr = "select SUM(sales_val) Sales,"
selstr = selstr & "month from sales_curr_year "
selstr = selstr & "group by month"

Set oradynaset = _
objdatabase.dbcreatedynaset(selstr, 0&)
```

```
...
...
...
endit:
exit sub

err_handler:
MsgBox _
"An unexpected error has been detected" & Chr(13) & _
"Description is: " & Err.Number & " , " & _
Err.Description
Resume endit
End sub
```

Listing 6. Using OO40-specific error-handling code.

```
Sub selectdata()

Dim objsession as object
Dim objdatabase as object

On error resume next

Set objsession = _
CreateObject("oracleinprocserver.xorasession")

if err.number <> 0 then
Msgbox "Unable to create object"
End
end if

Set objdatabase = _
objsession.opendatabase("mydb","myuser/mypass",0&)

if objsession.LastServerErr <> 0 Then
MsgBox objsession.LastServerErrText
objSession.LastServerErrReset
end
end if
```

```

selstr = "select SUM(sales_val) Sales,"
selstr = selstr & "month from sales_curr_year "
selstr = selstr & "group by month"

Set oradynaset = _
  ObjDatabase.dbcreatedynaset(selstr, 0&)

if objdatabase.LastServerErr <> 0 Then
  MsgBox objdatabase.LastServerErrText
  objdatabase.LastServerErrReset
end
end if
...
...
end sub

```

Summary

This article has shown that OO4O can be a powerful

technique for connecting your Oracle database to Windows client applications in general and Excel in particular. Not only that, but I'm sure you'd agree that compared to other methods available, it's also far simpler to code and maintain. For further information and examples of OO4O, pay a visit to Oracle's Web site. There you'll find dozens of useful links you can follow. Alternatively, just typing "OO4O" into a Web search engine such as Google will point you in the direction of many other good resources. ▲

Tom Reid lives and works in Edinburgh, Scotland. He develops Oracle systems and PC applications for a large investment bank. oracle_tips@hotmail.com.

Summer Reading...

Continued from page 6

easy, in fact, that you don't have to write it *at all*. This month's Download file contains a procedure called *genaa* (*genaa.sql*) that *generates* a caching package that emulates not only the primary key but also all of your unique indexes on that table.

I can generate and write out to file, for example, the summer reading package, by calling *genaa* as follows:

```

BEGIN
  genaa (
    tab_in => 'book',
    pkg_name_in => 'summer_reading',
    dir_in => d:\projects');
END;

```

I hope that this generator engine will make you more likely to take advantage of this functionality! ▲

 FEUER.ZIP at www.oracleprofessionalnewsletter.com

Steven Feuerstein is considered one of the world's leading experts on the Oracle PL/SQL language. He's the author or co-author of nine books on PL/SQL, including the now-classic *Oracle PL/SQL Programming* and *Oracle PL/SQL Best Practices* (all from O'Reilly & Associates). Steven is a Senior Technology Advisor with Quest Software, has been developing software since 1980, and worked for Oracle Corporation from 1987 to 1992. Steven is president of the Board of Directors of the Crossroads Fund, which makes grants to Chicagoland organizations working for social, racial, and economic justice (www.CrossroadsFund.org). steven.feuerstein@quest.com.

Don't miss another issue! Subscribe now and save!

Subscribe to *Oracle Professional* today and receive a special one-year introductory rate:
Just \$179* for 12 issues (that's \$20 off the regular rate)

NAME _____

COMPANY _____

ADDRESS _____

CITY STATE/PROVINCE ZIP/POSTAL CODE _____

COUNTRY IF OTHER THAN U.S. _____

E-MAIL _____

PHONE (IN CASE WE HAVE A QUESTION ABOUT YOUR ORDER) _____

- Check enclosed (payable to Pinnacle Publishing)
- Purchase order (in U.S. and Canada only); mail or fax copy
- Bill me later
- Credit card: __ VISA __ MasterCard __ American Express

CARD NUMBER _____ EXP. DATE _____

SIGNATURE (REQUIRED FOR CARD ORDERS) _____

Detach and return to:
Pinnacle Publishing ▲ 316 N. Michigan Ave. ▲ Chicago, IL 60601
Or fax to 312-960-4106

* Outside the U.S. add \$30. Orders payable in U.S. funds drawn on a U.S. or Canadian bank.



Pinnacle, A Division of Lawrence Ragan Communications, Inc. ▲ 800-493-4867 x.4209 or 312-960-4100 ▲ Fax 312-960-4106

Tip: Forcing Log Switches

Parin Jhaveri

Oracle9i has a very useful new feature that hasn't been discovered by many DBAs. Consider a disaster recovery scenario that requires a standby database. We need to establish a procedure to transfer archived logs from the production node to the standby node. Logs are transferred after each redo log is filled and a log switch occurs. However, at times of low activity, this may not occur for several hours. We need a method that forces a log switch on a regular basis—say, every 30 minutes—to ensure that the standby database is no more than 30 minutes behind the production database.

In Oracle8i, the only way to achieve this functionality is by scheduling a batch job via the DBMS_JOB supplied package or shell script to switch logs every 30 minutes. In Oracle9i, you can force the log writer to switch logs using the ARCHIVE_LAG_TARGET database initialization parameter. This parameter specifies the maximum number of seconds between each log switch. Thus, in our example, the parameter would be set

as follows (note that the parameter value is denominated in seconds):

```
ARCHIVE_LAG_TARGET = 1800
```

Note that because ARCHIVE_LAG_TARGET also considers estimated archival time and the existence of redo data in current redo logs, the time specified isn't the exact log switch time. Setting ARCHIVE_LAG_TARGET to a very low value can cause frequent log switches and hence frequent checkpoints, which could in turn degrade database performance.

Finally, you can disable time-based log switching by setting ARCHIVE_LAG_TARGET to 0, which happens to be the default setting. ▲

Parin Jhaveri is Senior Oracle DBA at Pacific Maritime Association, San Francisco. He has more than 10 years of experience working with Oracle products. pareen@yahoo.com.

September 2003 Downloads

- **FEUER.ZIP**—Source code to accompany Steven Feuerstein's article, "Summer Reading, Data Caching,

and String-Indexed Arrays."

For access to all current and archive content and source code, log in at www.oracleprofessionalnewsletter.com and enter the User name and Password at right when prompted.

User name

Password

Editor: Garry Chan (gchan@procaseconsulting.com)
CEO & Publisher: Mark Ragan
Group Publisher: Connie Austin
Executive Editor: Farion Grove
Production Editor: Andrew McMillan

Questions?

Customer Service:

Phone: 800-493-4867 x.4209 or 312-960-4100
Fax: 312-960-4106
Email: PinPub@Ragan.com

Editorial: FarionG@Ragan.com

Pinnacle Web Site: www.pinnaclepublishing.com

Subscription rates

United States: One year (12 issues): \$199; two years (24 issues): \$348
Other:* One year: \$229; two years: \$408

Single issue rate:

\$27.50 (\$32.50 outside United States)*

* Funds must be in U.S. currency.

Oracle Professional (ISSN 1525-1756)
is published monthly (12 times per year) by:

Pinnacle, A Division of Lawrence Ragan Communications, Inc.
316 N. Michigan Ave., Suite 400
Chicago, IL 60601

POSTMASTER: Send address changes to Lawrence Ragan Communications, Inc., 316 N. Michigan Ave., Suite 400, Chicago, IL 60601.

Copyright © 2003 by Lawrence Ragan Communications, Inc. All rights reserved. No part of this periodical may be used or reproduced in any fashion whatsoever (except in the case of brief quotations embodied in critical articles and reviews) without the prior written consent of Lawrence Ragan Communications, Inc. Printed in the United States of America.

Oracle, Oracle 8i, Oracle 9i, PL/SQL, and SQL*Plus are trademarks or registered trademarks of Oracle Corporation. Other brand and product names are trademarks or registered trademarks of their respective holders. *Oracle Professional* is an independent publication not affiliated with Oracle Corporation. Oracle Corporation is not responsible in any way for the editorial policy or other contents of the publication.

This publication is intended as a general guide. It covers a highly technical and complex subject and should not be used for making decisions concerning specific products or applications. This publication is sold as is, without warranty of any kind, either express or implied, respecting the contents of this publication, including but not limited to implied warranties for the publication, performance, quality, merchantability, or fitness for any particular purpose. Lawrence Ragan Communications, Inc., shall not be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this publication. Articles published in *Oracle Professional* reflect the views of their authors; they may or may not reflect the view of Lawrence Ragan Communications, Inc. Inclusion of advertising inserts does not constitute an endorsement by Lawrence Ragan Communications, Inc., or *Oracle Professional*.