NEW PL/SQL FEATURES IN ACTION: REAL WORLD CASE STUDIES

Bryn Llewellyn, PL/SQL Product Manager, Oracle Corp,
Steve Picciano, Product Development Manager, 170 Systems, Inc,
Brian Kuder, Application Development Engineer, 170 Systems, Inc.
Delivered as paper #32501 at OracleWorld, San Francisco, Mon 11-Nov-2002

INTRODUCTION
Both Oracle9i Database Release 1 and Release 2 introduced powerful new PL/SQL features. In this paper, we focus on just two: Native compilation of PL/SQL; and index-by-varchar2 tables. We will review the details of these enhancements and present real world case studies of their use by 170 Systems Inc and by Oracle Corporation’s own Applications Division. We will also list the Oracle9i Database new PL/SQL features for which we don’t give case studies in this paper.

SUMMARY OF ENHANCEMENTS
The new PL/SQL features in Oracle9i Database Release 1 were described in Paper 129 delivered at Oracle OpenWorld, San Francisco, Tue 4-Dec-2001, and the new PL/SQL features in Oracle9i Database Release 2 were described in Paper 30720 at OracleWorld, Copenhagen, Tue 25-June-2002. These are posted on the Oracle Technology Network website at...

otn.oracle.com/tech/pl_sql

The Release 1 new features are…

• Native compilation of PL/SQL
• CASE statements and CASE expressions
• Bulk binding enhancements: exception handling with bulk binding; bulk binding in native dynamic SQL
• Table functions and cursor expressions
• Multilevel collections
• Enhancements to the Utl_Http supplied package and various other packages
• Support for new SQL features (eg the MERGE statement)
• New object oriented features (support for subtypes)
• Various transparent enhancements (especially the introduction of the common SQL parser)

The Release 2 new features are…

• index-by-varchar2 tables (aka associative arrays)
• the ability to use RECORDs in DML and in BULK SELECTs
• enhancements to the Utl_File supplied package
support for the industry standard JDWP protocol to enable a tools vendor to build a PL/SQL debugger. This has already been exploited by Oracle’s JDeveloper in its Version 9.0.3.

Taken together, these enhancements make your PL/SQL applications more performant, enable new functionality, and give you new language constructs and tools to improve programmer productivity.

1. NATIVE COMPILATION OF PL/SQL

1.1. OVERVIEW

PL/SQL is often used as a thin wrapper for executing SQL statements, setting bind variables and handling result sets. In such cases the execution speed of the PL/SQL code is rarely an issue. It is the execution speed of the SQL that determines the performance. The efficiency of the context switch between the PL/SQL and the SQL operating environments might be an issue, but that's a different discussion. This is addressed by for example bulk binding and table functions.

However, we see an increasing trend to use PL/SQL for computationally intensive database independent tasks. It is after all a fully functional 3GL. Here it is the execution speed of the PL/SQL code that determines the performance. In pre-Oracle9i versions, compilation of PL/SQL source code always results in a representation (usually referred to as m-code or just bytecode) which is stored in the database and interpreted by the PL/SQL virtual machine which is part of the Oracle executable on the given platform. Oracle9i introduces a new approach. PL/SQL source code may optionally be compiled into native object code (shared libraries) which are dynamically linked into ORACLE.

We have measured a reduction in elapsed time for a computationally intensive unit (in single user tests) of about 40% when it’s compiled Native. While for data intensive programs native compilation may give only a marginal performance improvement, we have never seen it give performance degradation.

Furthermore, the m-code is loaded into the SGA whereas the dynamically linkable shared library is loaded into regular operating system memory. Thus when the whole database is compiled natively, a higher throughput may be possible due to reduced SGA contention.

1.2. ONE-TIME DBA SETUP

Native PL/SQL compilation is achieved by translating the PL/SQL source code into C source code which is then compiled on the given platform. The compiling and linking of the generated C source code is done using 3rd party utilities (see the platform specific documentation) whose location has been specified by the DBA via initialization parameters. The DBA should ensure that all these utilities are owned by the ORACLE owner (or a correspondingly trusted system user) and that only this user has write access to them. The dynamically linkable library for each natively compiled PL/SQL library unit is stored on the platform’s filesystem in directories, similarly under the DBA’s control. Thus native compilation does take longer than interpreted mode compilation. Our tests have shown a factor of about times two. This is because it involves these extra steps: generating C code from the initial output of the PL/SQL compilation; writing this to the filesystem; invoking and running the C compiler; and linking the resulting object code into ORACLE.

Oracle recommends that the C compiler is configured to do only basic optimization. Our tests have shown that optimizing the generated C beyond this produces negligible improvement in run-time performance but substantially increases the compilation time.

1.3. FREQUENTLY ASKED QUESTIONS

**WHAT OBJECTS ARE THE SUBJECT OF PL/SQL COMPILATION?**

```
select distinct object_type from dba_stored_settings
  where param_name = 'plsql_compiler_flags';
```

**FUNCTION**

**PACKAGE**
**PACKAGE BODY**

**PROCEDURE**

**TRIGGER**

**TYPE**

**TYPE BODY**

**WHAT’S THE GRANULARITY OF CHOICE FOR NATIVE VS INTERPRETED?**

It's the individual object, with the provision that a body (package or type) must be compiled in the same mode as it's spec.

**HOW DO I KNOW HOW A GIVEN UNIT IS COMPILED?**

```sql
select param_value from user_stored_settings
  where param_name = 'plsql_compiler_flags'
  and object_name = :MY_OBJECT
  and object_type = :MY_TYPE;
```

NATIVE, NON_DEBUG

The mode that’s stored with the library unit’s metadata (as revealed by this query) is used if the unit is implicitly recompiled as a consequence of dependency checking.

Note however that `Dbms_Utility.Compile_Schema` uses the current value of `plsql_compiler_flags` (see below) rather than the stored compilation mode.

**HOW DO I EXPRESS MY CHOICE?**

Via the initialization parameter `plsql_compiler_flags`. This can be set in the usual ways (`init.ora` and/or `alter system` with spfile persistence). It can also be set by `alter session`.

```sql
alter session set plsql_compiler_flags = 'NATIVE';
create or replace type My_Type as object (
  name varchar2(20),
  member procedure Show );
/
alter session set plsql_compiler_flags = 'INTERPRETED';
create or replace type body My_Type is
  member procedure Show
  is
    begin
      Dbms_Output.Put_Line ( name );
      end Show;
  end;
/
```

**PLS-00724: package body cannot be compiled to bytecode if its spec was compiled native**

...as explained.

**DOES ORACLE CORP RECOMMEND MIXING NATIVE AND INTERPRETED UNITS?**

No, because you might compromise some of your potential performance improvement.

Though mixing is supported (and never results in wrong behavior), we don't recommend mixing for your ultimate production system. Oracle Corporation recommends that all the PL/SQL library units that are called from a given top-level unit are compiled in the same mode. This is because there is a cost when a NATIVE unit calls an INTERPRETED one. Significantly, this recommendation includes the Oracle-supplied library units. (These are...
HOW DO I GET A 100% NATIVE DATABASE (OR 100% INTERPRETED)?

The simplest way to honor the recommendation above (Oracle recommends that all the PL/SQL library units that are called from a given top-level unit are compiled in the same mode) is to compile every PL/SQL library unit in the whole database NATIVE. A release soon after Oracle9i Database Version 9.2.0 will include such a script together with its partner to compile every PL/SQL library unit in the whole database INTERPRETED. Meanwhile, these are posted on OTN here…

http://otn.oracle.com/tech/pl_sql/htdocs/README_2188517.htm

DOES THIS MEAN THE ORACLE SHIPPED STUFF LIKE DBMS_OUTPUT WILL BE NATIVE TOO?

Yes of course. And of course we’ve tested this! In as much as lots of the shipped stuff is just a PL/SQL API for a C implementation, NATIVE won’t give a first order performance benefit for such units. But it will avoid the penalty of NATIVE calling INTERPRETED. (And anyway such units won’t take long to compile.)

ARE THE SHARED DYNAMICALLY LINKABLE LIBRARIES PORTABLE?

No! Absolutely not. Of course they could never be portable from platform to platform. But even in the “unofficial” case of TAR-ing up a whole Oracle Home and database on Solaris and un-TAR-ing it on a different Solaris machine you’ll find that the system is broken if you have NATIVE PL/SQL units. (This is due to second-order effects like how the system knows the path to a given .so.) Thus the only way you can achieve a NATIVE target installation is to explicitly compile the units NATIVE at the target.

DO I NEED A C COMPILING ENVIRONMENT AT THE DEPLOYMENT SITE(s)?

Yes. Non-negotiably. You have to compile all deployed units locally at each deployment site as explained above. And should one be invalidated, then the C environment is again needed to recompile it.

2. INDEX-BY-VARCHAR2 TABLES

2.1. OVERVIEW

The functionality of PL/SQL index-by tables has been enhanced by adding two new possibilities: index-by-pls_integer; and index-by-varchar2. We’ve adopted the term associative array for all these variants of index-by tables in line with common usage in other 3GLs for similar functionality.

The introduction of index-by-pls_integer means that the older datatype binary_integer need no longer be used in any new code.

The introduction of index-by-varchar2 provides a compact and easy-to-program technique for scenarios where values need to be accessed by non-numeric key.

Previously, as the following examples show, the functionality would have been explicitly programmed, for example by using Dbms_Utility.Get_Hash_Value to provide index values for an index-by-binary_integer table.

2.2. SCENARIO

The requirement to look up a value via a unique non-numeric key is a generic computational problem. Of course the Oracle9i Database provides a solution with SQL and a B*-tree index. But there’s a set of scenarios where considerable performance improvement can be obtained by instead using an explicit PL/SQL implementation. This was true even
before the new features discussed in this paper were available. These scenarios are characterized by very frequent
lookup in a relatively small set of values, usually in connection with flattening a relational representation for reporting
or for UI presentation. In order not to complicate the following examples with distracting detail, we’ll use a simple
neutral scenario.

Suppose we have a set of English-French vocabulary pairs stored persistently in the obvious way in a schema-level
table...

```sql
select * from translations;
ENGLISH       FRENCH
-------------------- ----------
computer      ordinateur
tree          arbre
book          livre
cabbage       chou
country       pays
vehicle       voiture
garlic        ail
apple         pomme
desk          écrivains
furniture     meubles
...
```

Our task is to allow lookup from French to English, and to allow efficient addition of new vocabulary pairs. This is
abstracted as the Vocab package, thus...

```plaintext
package Vocab is
    function Lookup ( p_english in varchar2 ) return varchar2;
    procedure New_Pair ( p_english in varchar2, p_french in varchar2 );
end Vocab;

And we’ll use this test...

begin
    Dbms_Output.Put_Line ( Vocab.Lookup ( 'tree' ) );
    Vocab.New_Pair ( 'garden', 'jardin' );
    Dbms_Output.Put_Line ( Vocab.Lookup ( 'garden' ) );
end;
```

Of course all implementations satisfy the test. We’ll examine some alternative implementations, leaving the new – and
best – approach to last.

### 2.3. NAÏVE PURE SQL APPROACH

Here we’re concerned only about correctness of behavior and ease of algorithm design. We’ll accept the performance
we get.

```plaintext
package body Vocab is
    function Lookup ( p_english in varchar2 ) return varchar2
    is
        v_french translations.french%type;
    begin
        select french into v_french from translations
        where english = p_english;
        return v_french;
    end Lookup;
```
procedure New_Pair ( p_english in varchar2, p_french in varchar2 ) is
begin
  insert into translations ( english, french ) values ( p_english, p_french );
end New_Pair;
end Vocab;

The algorithm is of course trivial and is due entirely to native SQL language features. Each time Lookup is invoked, we make a round trip between PL/SQL and SQL. The associated expense of this frequently repeated context switch could become very significant.

2.4. LINEAR SEARCH IN INDEX-BY-BINARY_INTEGER TABLE

Here we recognize that we can improve performance by eliminating the round trip between PL/SQL and SQL for lookup by caching all the target values in an appropriate PL/SQL structure. Pre Oracle9i Database Release 2, the natural choice is an index-by table, since we are happy with the persistent representation as a relational table and do not want to maintain any persistent denormalization.

*Note:* the following implementation was coded to run in Oracle9i Database Version 9.0.1. It also runs in Oracle8i Database Version 8.1.7.

```plsql
package body Vocab is
type word_list is table of translations%rowtype
  index by binary_integer /* can't use pls_integer pre-9.2 */;
g_english_french word_list;

function Lookup ( p_english in varchar2 ) return varchar2
is
begin
  for j in 1..g_english_french.Last()
    loop
    if g_english_french(j).english = p_english
      then
        return g_english_french(j).french;
      end if;
    end loop;
end Lookup;

procedure New_Pair ( p_english in varchar2, p_french in varchar2 ) is
idx binary_integer;
begin
  idx := g_english_french.Last() + 1;
g_english_french(idx).english := p_english;
g_english_french(idx).french := p_french;
insert into translations ( english, french ) values ( p_english, p_french );
end New_Pair;
begin /* package initialization */
declare idx binary_integer := 0;
begin
  for j in ( select english, french from translations )
    loop
    idx := idx+1;
    g_english_french(idx).english := j.english;
g_english_french(idx).french := j.french;
  end loop;
end;
end Vocab;
```
The algorithm is still trivial, but does require some explicit coding.

The entire table contents are loaded into a PL/SQL `index-by-binary_integer` table when the package is initialized. Pre-Version 9.2.0, we cannot use the more efficient and compactly written `BULK` construct to initialize the `index-by` table. The attempt causes compilation error `PLS-00597` in Version 9.0.1 and earlier. This restriction is lifted in Oracle9i Database Release 2 enabling us to write the initialization thus...

```plsql
declare cursor cur is select english, french from translations;
begin
  open cur; fetch cur bulk collect into g_english_french; close cur;
end;
```

### 2.5. HASH-BASED LOOKUP IN INDEX-BY-BINARY_INTEGER TABLE

The linear search algorithm suffers from the well-known disadvantage that on average we’ll examine half the elements in the `index-by` table before we find a match. A possible improvement is to maintain the elements in lexical sort order and to use a binary chop algorithm: compare the search target to the half-way element to determine which half it’s in; repeat this test recursively on the relevant half. This requires more elaborate coding – and testing – and poses a design problem for the `New_Pair` procedure. Either the array must be opened at the insertion point copying all later elements to the next slot, or it must be created sparse. Neither of these approaches is very comfortable, and the latter is not complete until the corner case where a gap fills up is catered for.

A more popular approach therefore is to determine the element index for storing the current value by hashing that value.

*Note*: the following implementation was coded to run in Oracle9i Database Version 9.0.1. It also runs in Oracle8i Database Version 8.1.7.

```plsql
package body Vocab is
  /* can't use pls_integer pre-9.2 */
  hash                 binary_integer;
  g_hash_base constant number := 100 /* care must be given to... */;
  g_hash_size constant number := 100 /* ...the choice of these values */;
  type word_list is table of translations.french%type
                  index by binary_integer;
  g_english_french word_list;

  function Lookup ( p_english in varchar2 ) return varchar2 is
    begin
      hash := Dbms_Utility.Get_Hash_Value ( p_english, g_hash_base, g_hash_size );
      return g_english_french(hash);
    end Lookup;

  procedure New_Pair ( p_english in varchar2, p_french in varchar2 ) is
    begin
      hash := Dbms_Utility.Get_Hash_Value ( p_english, g_hash_base, g_hash_size );
      g_english_french(hash) := p_french;
      insert into translations ( english, french ) values ( p_english, p_french );
    end New_Pair;

  begin /* package initialization */
    begin
      for j in ( select english, french from translations )
        loop
          hash := Dbms_Utility.Get_Hash_Value ( j.english, g_hash_base, g_hash_size );
          g_english_french(hash) := j.french;
        end loop;
      end;
    end Vocab;
```
The above algorithm is too naïve for real-world use. There is no guarantee that two distinct values for the
name IN parameter to Get_Hash_Value will always produce distinct hash values. Thus to be robust, a collision avoidance
scheme must be implemented. This is a non-trivial design, implementation and testing task. (Oracle provides no
specific support for this.)

It's probable that the resulting index-by-binary_integer table will be quite sparse. This point will be discussed after
describing the new index-by-varchar2 table.

### 2.6. Direct Lookup in Index-by-varchar2 Table

Of course a yet more elaborate ambition pre-Oracle9i Database Release 2 would be – after studying the relevant
computer science textbooks – to implement a B*-tree structure in PL/SQL, horror of wheel re-invention
notwithstanding!

The datastructure might this…

```plsql
type Node_t is record (
  value varchar2(20),
  left_child binary_integer /* refer to the array index... */,
  right_child binary_integer /* ...of the... */,
  parent binary_integer /* ...relevant element. */);
type Btree_t is table of Node_t index by binary_integer;
the_tree Btree_t;
```

However, the implementation would be very far from trivial, and is certainly too long and complex for inclusion in
this paper.

A far better approach – newly possible in Oracle9i Database Release 2 – is to use precisely the B*-tree organization of
the values but to do so implicitly via a language feature, the index-by-varchar2 table.

You can think of the index-by-varchar2 table as the in-memory PL/SQL version of the schema-level index organized
table. Here’s the syntax…

```plsql
declare
  idx varchar2(20);
type word_list is table of varchar2(20)
    index by idx%type;
the_list word_list;
begin
  the_list ('computer') := 'ordinateur';
  ...  
  the_list ('tree')     := 'arbre';
/* loop in lexical sort order */
  idx := the_list.First();
  while idx Is not null 
    loop
      Dbms_Output.Put_line ( idx || ' : ' || the_list(idx) );
      idx := the_list.Next(idx);
    end loop;
end;
```

The index-by-varchar2 table is optimized for efficiency of lookup on a non-numeric key, where the notion of sparseness
is not really applicable. The index-by-_integer table (where now _integer can be either pls_integer or binary_integer) in
contrast is optimized for compactness of storage on the assumption that the data is dense.

This implies that there might be cases where, even though the key is inherently numeric, it’s better to represent it as a
index-by-varchar2 table via a To_Char conversion.
This new construct now allows the following very natural implementation of the `Vocab` body…

```sql
package body Vocab is

type word_list is table of translations.french%type
  index by translations.english%type;

  g_english_french word_list;

  function Lookup ( p_english in varchar2 ) return varchar2
  is
    return g_english_french( p_english );
  end Lookup;

  procedure New_Pair ( p_english in varchar2, p_french in varchar2 ) is
    g_english_french( p_english ) := p_french;
    insert into translations ( english, french ) values ( p_english, p_french );
  end New_Pair;

begin /* package initialization */
  for j in ( select english, french from translations )
  loop
    g_english_french( j.english ) := j.french;
  end loop;
end Vocab;
```

Note that it is not yet possible to use a `index-by-varchar2` table as the target or source of a bulk binding construct.

The body of function `Lookup` is now so trivial that you might decide to dispense with it altogether (suppressing a mild qualm about violating some rule of best practice) and use this stripped down implementation…

```sql
package Vocab is

type word_list is table of translations.french%type
  index by translations.english%type;

  lookup word_list;

  procedure New_Pair ( ... );
end Vocab;
```

```sql
package body Vocab is

  procedure New_Pair ... end New_Pair;
begin /* package initialization */
  for j in ( select english, french from translations )
  loop
    lookup ( j.english ) := j.french;
  end loop;
end Vocab;
```

2.7. PERFORMANCE COMPARISON

The four different approaches shown above were timed (elapsed time, single user on a quiet high-end laptop) using a dataset of 1 million translation pairs generated thus…

```sql
for indx in 1 .. 1000000 loop
  insert into translations
    values ('english' || indx, 'french' || indx);
end loop;
```

…by running varying numbers of translations thus…
v := /* keep the initialization out of the timing */
    Vocab.Lookup ('english1');
    t0 := Dbms_Utility.Get_Time();
    for indx in 1 .. p_iterations
    loop
        v := Vocab.Lookup ('english' || indx);
    end loop;
    t1 := Dbms_Utility.Get_Time();

For interest, the pure SQL method was run both with and without a supporting B*-tree index on the looked-up
english column.
The times are in milliseconds per 1000 lookups.

Table 1: Comparing the approaches in an INTERPRETED environment.

<table>
<thead>
<tr>
<th>Number of iterations</th>
<th>Pure SQL without B*-tree index</th>
<th>Pure SQL with B*-tree index</th>
<th>Linear scan in index-by-binary_integer table</th>
<th>Hash lookup in index-by-binary_integer table</th>
<th>Direct lookup in index-by-varchar2 table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>650</td>
<td>190</td>
<td>750</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>5,000</td>
<td>3286</td>
<td>164</td>
<td>4956</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>10,000</td>
<td>6372</td>
<td>157</td>
<td>10044</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>50,000</td>
<td>156</td>
<td>17</td>
<td>18</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>100,000</td>
<td>166</td>
<td>18</td>
<td>18</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>500,000</td>
<td>162</td>
<td>17</td>
<td>18</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1,000,000</td>
<td>187</td>
<td>18</td>
<td>18</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 2: Comparing the approaches in an NATIVE environment.

<table>
<thead>
<tr>
<th>Number of iterations</th>
<th>Pure SQL without B*-tree index</th>
<th>Pure SQL with B*-tree index</th>
<th>Linear search in index-by-binary_integer table</th>
<th>Hash lookup in index-by-binary_integer table</th>
<th>Direct lookup in index-by-varchar2 table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>620</td>
<td>180</td>
<td>470</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>5,000</td>
<td>3088</td>
<td>132</td>
<td>3424</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>10,000</td>
<td>6303</td>
<td>144</td>
<td>7106</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>50,000</td>
<td>150</td>
<td>17</td>
<td>18</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>100,000</td>
<td>144</td>
<td>17</td>
<td>18</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>500,000</td>
<td>155</td>
<td>18</td>
<td>18</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1,000,000</td>
<td>158</td>
<td>17</td>
<td>17</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

We see that the times for the index-supported methods scale linearly (ie the average time per lookup is fairly constant). But the times for the linear scan methods scale very badly. We’d expect the average time for such scans to scale as the “sum of 1-to-N” for N elements.
The index-by-varchar2 PL/SQL table is faster than the index schema-level table by a factor of about 20 times. In these tests, the index-by-varchar2 table is faster than the hash lookup in the index-by-binary_integer table by a factor of about 2 times. Note though that this test unfairly favors the hash lookup because it was written without any collision
avoidance.
Interestingly, because the linear search in index-by-binary_integer table method involves most PL/SQL instructions, it benefits most from NATIVE compilation. Here we see the 30-40% improvement which characterizes computationally intensive program units. But the greater benefit derives from using an algorithm which cuts down on the number of PL/SQL instructions. The direct lookup in index-by varchar2 table method is so efficient that the effect of benefit of NATIVE compilation cannot be detected!

2.8. USING AN INDEX-BY-VARCHAR2 TABLE FOR SORTING

The discussion above focused on the use of the the index-by-varchar2 table to support lookup. It also provides implicit functionality to sort items based on lexical sorting order. You just insert your elements in any order into the table and then traverse it like this…

```
DECLARE
    runner varchar2(20);
    type items_t is table of boolean /* arbitrary choice of type */
        index by runner%type;
    sorted_items items_t;
BEGIN
    sorted_items ( 'zebra' ) := true;
    sorted_items ( 'ape'   ) := true;
    sorted_items ( 'cat'   ) := true;
    ...
    runner := sorted_items.First();
    WHILE runner is not null LOOP
        Show ( runner );
        runner := sorted_items.Next(runner);
    END LOOP;
END;
```

In this example we never access what is stored in the table, so its type is unimportant. Boolean was chosen arbitrarily. We’re interested only in the index of the table.

2.9. INDEX-BY-VARCHAR2 TABLES: SUMMARY

The figures above vividly illustrate the well-known benefits of an index-based lookup over a linear scan. They also show that the PL/SQL structure is about twenty times faster than the corresponding schema-level structure, at least up to a million elements. However, before the introduction of the index-by-varchar2 table, the programming task required to avoid linear scans in PL/SQL tables was rather daunting.

The new ability to use an index-by-varchar2 table is a major enhancement that allows a class of very common programming tasks to be implemented much more performantly than was possible pre Version 9.2.0, with very much less design and testing, and in dramatically fewer lines of code.

3. CASE STUDY: SIMULATED PAYROLL RUN

3.1. SCENARIO

All the code required to run this scenario is presented in Appendix A.1.

We consider a data model which is a little more sophisticated than the shipped HR demo schema. In our scenario, each employee has a job title (just as in the shipped schema) but then each job title has a salary code which refers to a list of salaries. So the salary for an employee needs to be looked up through two levels of indirection thus…
The simulated payroll run simply generates a list of each employee with his looked up salary. The performance of two methods is compared.

### 3.2. IMPLEMENTATION

The SQL method (encapsulated in procedure `Payroll.By_Sql`) runs a cursor over a three-table join…

```sql
select last_name, salary
from
  employees
 inner join jobs using (job_id)
 inner join salaries using (sal_code)
```

The PL/SQL method (encapsulated in procedure `Payroll.By_Plsql`) runs a cursor over a single table…

```sql
select last_name, job_id from employees
...and for each row it looks up the salary in an index-by-varchar2 table thus…

v_salary := v_sal_tab(this_employee.job_id)
...where the index-by-varchar2 table is populated at the start of the run thus…

for j in ( select job_id, salary
  from jobs inner join salaries using (sal_code) )
  loop v_sal_tab(j.job_id) := j.salary; end loop;
```

Each procedure also calculates the grand total of all salaries as a cross-check for correctness of the algorithms.

The data in the shipped HR schema was used as the starting point. An extra column `sal_code` was added to the jobs table and populated with the distinct values President, Executive, Manager, Staff. An new table `salaries` was created as the reference table for jobs.sal_code with appropriate salary values. The shipped content in the employees table was replicated 2000 times to give a total row count of about 210 thousand.

The timing was done by procedure `Payroll.Time`. This calls each of `By_Sql` and `By_Plsql` before starting the clock to avoid timing anomalous start-up effects. It then loops five times and calls the two procedures in succession in each iteration, recording the elapse time for each simulated payroll run with `Dbms_Utility.Get_Time`.

### 3.3. PERFORMANCE COMPARISON EXPERIMENT

Significantly, the performance of `By_Sql` will depend critically on the execution plan for the three-table join, while the performance of `By_Plsql` is insensitive to the choice of plan (the only way to traverse the single table of employees is a table scan, and the select statement that populates the index-by-varchar2 table runs in immeasurable short time (the large jobs table has only 19 rows). Thus the timings for `By_Sql` were recorded for two different plans (by using hints)…

**Alt 1:**
SELECT STATEMENT
NESTED LOOPS

NESTED LOOPS
  TABLE ACCESS FULL JOBS
  TABLE ACCESS BY INDEX ROWID SALARIES
  INDEX UNIQUE SCAN SALARIES_SAL_CODE
  TABLE ACCESS FULL EMPLOYEES

Alt 2:
SELECT STATEMENT
  HASH JOIN
    TABLE ACCESS FULL JOBS
    TABLE ACCESS FULL SALARIES
    TABLE ACCESS FULL EMPLOYEES

The experiment was done both with the Payroll package compiled NATIVE and with it compiled INTERPRETED. Not surprisingly (see Section 2.7) the times were not dramatically different: NATIVE was about 1-2% faster.

Table 5: Comparing the times (in seconds) for the simulated payroll run. Only the NATIVE times are shown

<table>
<thead>
<tr>
<th>By_Sql (Nested Loops)</th>
<th>By_Sql (Hash Join)</th>
<th>By_Plsql</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.79</td>
<td>7.92</td>
<td>7.53</td>
</tr>
<tr>
<td>12.76</td>
<td>7.86</td>
<td>7.52</td>
</tr>
<tr>
<td>12.85</td>
<td>7.91</td>
<td>7.63</td>
</tr>
<tr>
<td>12.80</td>
<td>8.27</td>
<td>7.65</td>
</tr>
<tr>
<td>12.79</td>
<td>8.76</td>
<td>7.98</td>
</tr>
<tr>
<td>12.79 (avg)</td>
<td>8.14 (avg)</td>
<td>7.66 (avg)</td>
</tr>
</tbody>
</table>

Note: the Hash Join plan is clearly the better alternative, and when the tables have been analyzed as is recommended, the optimizer does choose this plan. When the tables have not been analyzed (as is of course not recommended) the optimizer chooses the Nested Loops plan. The tests with run with no statistics to allow the use of hints to determine the plan.

3.4 INTERPRETATION

How should we interpret these results in comparison with those presented in Section 2.7 for the vocabulary lookup scenario? There we considered a design problem which arose in the context of a PL/SQL program unit. We needed to perform a lookup on a varchar2 key and saw that it was faster by a factor of about 20 to remain within the PL/SQL engine and use an index-by-varchar2 table than it was to make the context switch to the SQL engine for each lookup (though of course the SQL access by B*-tree index is intrinsically very fast).

In the payroll scenario, our design problem arises in the context of demoralizing relational data stored in fully normalized form for the purpose of display. We can perform the denormalization either in the SQL engine or the PL/SQL engine, and in each case we remain in the chosen environment without repeated context shift.

In this experiment, the approach that implements the denormalization with a PL/SQL index-by-varchar2 table is about 10% faster than the approach that implements it with a SQL join.

4. CASE STUDY: PERFECT TRIANGLES CALCULATION (REVISITED)

This scenario was introduced in Paper 129 delivered at Oracle OpenWorld, San Francisco, Tue 4-Dec-2001 to illustrate the benefits of Native compilation. It also provides a nice vehicle to show the performance improvements
due to index-by-varchar2 tables.

The problem (find all right-angled triangles with all side lengths integer a.k.a. perfect triangles) is described in Appendix A.2. The algorithm depends on exhaustively testing all possible candidates up to a specified maximum side length and storing each that satisfies the Pythagoras theorem test. The twist is that having found the, say, 3,4,5 triangle we must not count its integral multiples, 6,8,10 and so on. The duplicate avoidance is supported by populating a list of unique triangles found to date with their generated multiples. Each new candidate must be checked against every member in this list for uniqueness.

Three different approaches to implementing the list of current unique solutions were timed (on a high-end laptop running Version 9.2.0 with no load apart from the test), both when compiled Interpreted and when compiled Native. The code is listed in Appendix A.2, and the times are given below.

<table>
<thead>
<tr>
<th></th>
<th>Interpreted compilation</th>
<th>Native compilation</th>
<th>Percent speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear scan in PL/SQL</td>
<td>118.8</td>
<td>86.8</td>
<td>37</td>
</tr>
<tr>
<td>index by pls_integer table</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indexed lookup in schema-level global temporary table</td>
<td>76.6</td>
<td>65.0</td>
<td>18</td>
</tr>
<tr>
<td>Indexed lookup in PL/SQL index-by-varchar2 table</td>
<td>73.7</td>
<td>61.9</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Interpreted compilation</th>
<th>Native compilation</th>
<th>Percent speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear scan in PL/SQL</td>
<td>188.9</td>
<td>147.0</td>
<td>29</td>
</tr>
<tr>
<td>index by pls_integer table</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indexed lookup in schema-level global temporary table</td>
<td>123.4</td>
<td>100.5</td>
<td>23</td>
</tr>
<tr>
<td>Indexed lookup in PL/SQL index-by-varchar2 table</td>
<td>113.1</td>
<td>96.1</td>
<td>18</td>
</tr>
</tbody>
</table>

The speed increase due to NATIVE compilation is in the range of about 20% to 40%, depending on the nature of the algorithm and the mix of SQL and PL/SQL. We see again the danger of the linear scan, and the superiority of the index-by-varchar2 PL/SQL table.

5. ORACLE9i IN ACTION: 170 SYSTEMS INC

This section highlights an example of how 170 Systems Inc. was able to quickly leverage two new PL/SQL features provided by the Oracle 9i Database with the primary goal of improving overall product performance and, ultimately, end-user performance for their customers.

5.1. ABOUT 170 SYSTEMS INC.

170 Systems, Inc (www.170Systems.com) has provided Content Management, Document Management, Document Imaging, Business Process Management software and services since 1990. 170 Systems is a close Oracle ISV Partner and has participated in the Beta Program for the Oracle9i Database Version 9.2.0. They have now certified their 170
The 170 MarkView Document Management and Imaging System provides Content Management, Document Management, and Imaging solutions - all tightly integrated with the Oracle9i Database, Oracle9i Application Server and the Oracle E-Business Suite. Enabling businesses to capture and manage all of their information online in a single, unified system - regardless of original source or format - the 170 MarkView solution provides scalable, secure, production-quality Internet B2B and intranet access to all of an organization's vital information, while streamlining the associated business processes and maximizing operational efficiency. The 170 MarkView solution extends and leverages the functionality of enterprise applications.

A large-scale multi-user, multi-access system, 170 MarkView supports the very large numbers of documents, images, concurrent users, and the high transaction rates required by 170 Systems customers. Therefore performance and scalability are especially important. 170 Systems customers include Global 1000 deployments in over 40 countries.

5.2. CASE STUDY: USING INDEX-BY-VARCHAR2 TABLES AND NATIVE COMPILATION

170 Systems are planning to take advantage of many of the new PL/SQL features that are available in Version 9.2.0, for example, index-by-varchar2 tables to improve performance in an application that performs complex stack manipulations. Values are taken from the stack using random access based on an identifying character name. In some cases, values were accessed in a processing intensive manner, therefore a performance improvement is expected by using index-by-varchar2 tables implemented natively in Version 9.2.0.

The product used for the case study is a web application that allows a developer to quickly create an end-user web application for displaying 170 MarkView document information in an organized manner. These custom web applications typically augment the financial services, human resources, or contract management process. Due to the nature of these applications, they require scalability to support global web applications that present complex web pages and large amounts of data and content. The application is a processing intensive PL/SQL-based engine that provides a common, re-usable infrastructure for developing and deploying the web applications.

Due to the complexity of this product, performance of the application code and the end-user web applications are a common concern. This application was a good candidate for using index-by-varchar2 tables and Native Compilation. The following section details some product code samples that were replaced with index-by-varchar2 tables.

**DATA STRUCTURE DECLARATION**

**Before…**

```sql
type Property_Record /* encapsulates properties */
    is record
    (property_name  varchar2(256),
     property_value long);

type Property_List is table of Property_Record
    index by binary_integer;

propertylist Property_List;
propertyIndex number := 0;
```

**After…**

```sql
type Property_List is table of long index by varchar2(259);
propertyList Property_List;
```

**ADDING A PROPERTY TO THE LIST**

**Before…**

```sql
procedure AddProperty (propertyName in varchar2,
                      propertyValue in long) is
```

**After…**

```sql
procedure AddProperty (propertyName in varchar2,
                      propertyValue in long) is
```
begin
  propertyList(propertyIndex + 1).property_name := propertyName;
  propertyList(propertyIndex + 1).property_value := propertyValue;
  propertyIndex := propertyIndex + 1;
end AddProperty;

After...
procedure AddProperty (  
  propertyName in varchar2,  
  propertyValue in long ) is
begin
  propertyList(upper(propertyName)) := propertyValue  
  /* add (or update) a property to the list */;
end AddProperty;

RETRIEVING A VARIABLE FROM THE LIST

Before...
function GetProperty (  
  propertyName in varchar2 ) return long is
begin
  for nextRow in reverse 1..propertyIndex  
  /* linear search from the most-recent element */ loop
    if propertyList(nextRow).property_name = upper(propertyName) then
      return propertyList(nextRow).property_value;
    end if;
  end loop;
  return null;
end GetProperty;

After...
function GetProperty (  
  propertyName in varchar2 ) return long is
begin
  if propertyList.exists(upper(propertyName)) then
    return propertyList(upper(propertyName));
  end if;
  return null;
end GetProperty;

Note the similarity with Vocab.New_Pair and Vocab.Lookup in Section 2 above. Here a general understanding of usage patterns leads to choosing to start the linear search from the most recently added element.

The typical population of propertyList is about 100-200 elements. It was hoped that due to this small size we could escape the effort of programming a hashing approach in the original index-by-binary_integer table implementation.

PERFORMANCE MEASUREMENTS

After completing the product changes, we selected a sample web application that was indicative of a typical web application deployed at a financial services customer. This application selects 170 MarkView document information based on a user-entered query. For this example we selected queries that would return document information for approximately 95 documents. The test simulated 100 trials, assuming a limited number of users.

The first test consisted of using Native Compilation to compile the product PL/SQL packages in the database. The typical performance improvement for this test scenario was approximately 20%.

The second test consisted of changing the product code to replace existing code with an implementation that included the use of index-by-varchar2 tables. These packages were compiled without using Native Compilation. The typical
The third test consisted of using Native Compilation to compile the new product code that included the use of index-by-varchar2 tables, combining these two features. The typical performance improvement for this test scenario was approximately 47%.

It's interesting that the improvements measure separately in the first and second tests don’t multiply when Native Compilation and index-by-varchar2 tables are used together. This is of course because the use of index-by-varchar2 tables eliminates very many PL/SQL operations, so there are then few left to benefit from the Native Compilation!

These three tests were repeated using a customer example with a very large data result set being processed by the processing intensive PL/SQL packages used in this application. This test achieved a performance improvement of 30% when using Native Compilation, a performance improvement of 90% when leveraging index-by-varchar2 tables, and a performance improvement of 91% when combining both features.

ADDITIONAL RESULTS

During the regression testing of the 170 MarkView product suite using the Oracle 9.0.1 Database release, we completed a performance analysis using Native Compilation. In some of the product code, we experienced a performance improvement up to 40%.

5.3. 170 SYSTEMS CASE STUDY: CONCLUSION

As a result of testing the index-by-varchar2 tables and Native Compilation features in the scope of our products, we implemented the index-by-varchar2 table code changes in the application and modified the installation scripts to compile the new Oracle9i compliant packages when installing into a Oracle 9.2.0 database. We also created installation scripts to optionally use Native Compilation when installing the PL/SQL packages for this application as well as other 170 Systems products.

6. ORACLE9i IN ACTION: ORACLE E-BUSINESS SUITE

The database component of the E-Business Suite has about 46 thousand PL/SQL objects, due mainly to packages and their bodies. There are about 23 thousand packages. (This number is growing; it was 24.5 at the last count!). This corresponds to about 20 million lines of PL/SQL source code.

The performance improvement due to Native Compilation was investigated using the standard concurrent programs Order Import, Payroll, and Auto Invoice.

6.1. HARDWARE SPECIFICATION

The testing was purposely carried out on a lower-end system in order to determine if the run-time performance improvement justified the increase in compilation time.

- **Server Class:** Sun Ultra E450
- **CPU Speed:** 296 MHz
- **Number of CPUs:** 4
- **Main Memory:** 4 GB
- **Operating System:** Solaris 2.6 with Patch 105181-33

On this system, the recompilation of the whole database took approx. 22hr for Native and approx. 8hr for Interpreted.

6.2. SOFTWARE VERSIONS AND CONFIGURATION

- Oracle Applications 11i Release 11.5.7
- Oracle9i Release 2, Version 9.2.0.1 (production release)
- Sun Workshop 6, Update 1
Sun Workshop C Compiler Version 5.2
The whole database was compiled Native, producing about 46 thousand .so files on a single directory (the plsql_native_library_subdir_count initialization parameter was set to zero).

6.3. RESULTS
Order Import is a PL/SQL intensive program. It was tested through the entire order to cash cycle. Its performance is characterized by the number of order lines processed per hour.
Payroll consists mainly of a PRO*C client. However, the PRO*C client makes calls to PL/SQL server side packages including fast formulas. Its performance is characterized by the number of assignments processed per hour.
Auto Invoice again consists mainly of a PRO*C client. Again, the PRO*C client makes calls to PL/SQL server side packages including the tax engine and auto accounting. Its performance is characterized by the number of invoice lines processed per hour.

<table>
<thead>
<tr>
<th>Table 6: Throughput in items per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order Import</td>
</tr>
<tr>
<td>Interpreted:</td>
</tr>
<tr>
<td>3,015</td>
</tr>
<tr>
<td>Native:</td>
</tr>
<tr>
<td>3,983</td>
</tr>
<tr>
<td>Improvement:</td>
</tr>
<tr>
<td>32%</td>
</tr>
<tr>
<td>Payroll:</td>
</tr>
<tr>
<td>Interpreted:</td>
</tr>
<tr>
<td>26,105</td>
</tr>
<tr>
<td>Native:</td>
</tr>
<tr>
<td>27,342</td>
</tr>
<tr>
<td>Improvement:</td>
</tr>
<tr>
<td>5%</td>
</tr>
<tr>
<td>Auto Invoice</td>
</tr>
<tr>
<td>Interpreted:</td>
</tr>
<tr>
<td>1,007,633</td>
</tr>
<tr>
<td>Native:</td>
</tr>
<tr>
<td>1,102,088</td>
</tr>
<tr>
<td>Improvement:</td>
</tr>
<tr>
<td>9%</td>
</tr>
</tbody>
</table>

The average CPU utilization was measure during the tests. It was 94% for the Interpreted case, and 89% for the Native case. The reduced CPU utilization in the Native case improves performance and scalability. In the Interpreted case, CPU utilization often reached 100%, and was continuously sporadic. In the Native case it was much more uniform and stable.

6.4. E-BUSINESS SUITE CASE STUDY: CONCLUSION
The overall performance improvement provided by Native PL/SQL Compilation varies from module to module depending on the amount of PL/SQL processing within the module. In some of the modules such as Order Import, a 30% performance improvement was observed while other modules resulted in less gain. Native Compilation is transparent to Oracle Applications including patching. When patches are installed, the PL/SQL unit will automatically be compiled as a native unit.
The results of this performance study demonstrate the value of the PL/SQL Native Compilation feature. Oracle Corporation recommends it to customers to improve performance and scalability of their E-Business Suite environment.

CONCLUSION
Native Compilation is a transparent feature which can boost performance of amenable applications.
Index-by-varchar2 tables are a new language feature. They provide you with a highly performant PL/SQL solution to access array data via a character key.
This paper has provided copy&pastable code for case studies to investigate the performance benefit of these features, and to understand the scenarios which will most benefit from them.
The performance gains illustrated in the 170 Systems and Oracle E-Business Suite case studies are in line with expectations set by the simple scenarios implemented in the copy&pastable code.
Using these features is easy, and we strongly encourage you to try them out, first with the copy&pastable code and then in your own applications.
**APPENDIX**

**A.1. CODE TO IMPLEMENT THE PAYROLL SCENARIO FROM SECTION 3**

**CREATE THE DATA**
Connect /as sysdba
drop user big_hr cascade;
create user big_hr identified by p;
grant resource, connect to big_hr;
grant select on hr.employees to big_hr;
grant select on hr.jobs to big_hr;

Connect big_hr/p
create table employees as
select
    employee_id,
    last_name,
    salary original_salary,
    job_id
from hr.employees;
update employees set employee_id = Rownum;
alter table employees modify (employee_id not null);
create unique index employees_employee_id on employees(employee_id);
alter table employees
    add primary key (employee_id)
    using index employees_employee_id;

create table jobs as
select
    job_id,
    job_title
from hr.jobs;
alter table jobs
    add sal_code varchar2(10);
create or replace view avg_sal_by_job_title as
select
    Avg(original_salary) avg_sal, job_title
from
    employees inner join jobs using (job_id)
    group by job_title;
update jobs set sal_code = 'President'
where job_title = 'President';
update jobs set sal_code = 'Executive'
where job_title in (  
    select job_title from avg_sal_by_job_title
    where avg_sal >= 10000 and job_title <> 'President');
update jobs set sal_code = 'Manager'
where job_title in (  
    select job_title from avg_sal_by_job_title
    where avg_sal >= 5000 and avg_sal < 10000 and job_title <> 'President');
update jobs set sal_code = 'Staff'
where job_title in (  
    select job_title from avg_sal_by_job_title


WHERE AVG_SAL < 5000 AND JOB_TITLE <> 'President');

SELECT
    SAL_CODE, AVG_SAL, JOB_TITLE
FROM
    JOBS INNER JOIN AVG_SAL_BY_JOB_TITLE USING (JOB_TITLE)
ORDER BY AVG_SAL DESC
/
President    24000 President
Executive    17000 Administration Vice President
Executive    13000 Marketing Manager
Executive    12200 Sales Manager
Executive    12000 Accounting Manager
Executive    12000 Finance Manager
Executive    11000 Purchasing Manager
Executive    10000 Public Relations Representative
Manager       8350 Sales Representative
Manager       8300 Public Accountant
Manager       7920 Accountant
Manager       7280 Stock Manager
Manager       6500 Human Resources Representative
Manager       6000 Marketing Representative
Manager       5760 Programmer
Staff         4400 Administration Assistant
Staff         3215 Shipping Clerk
Staff         2785 Stock Clerk
Staff         2780 Purchasing Clerk
*/;

CREATE OR REPLACE VIEW EMPLOYEES_WITH_SAL_CODES AS
SELECT
    LAST_NAME, ORIGINAL_SALARY, SAL_CODE
FROM
    EMPLOYEES INNER JOIN JOBS USING (JOB_ID);

CREATE TABLE SALARIES AS
SELECT SAL_CODE, FLOOR(AVG(ORIGINAL_SALARY)) SALARY
FROM EMPLOYEES_WITH_SAL_CODES
GROUP BY SAL_CODE;

SELECT SAL_CODE, SALARY
FROM SALARIES
ORDER BY SALARY DESC
/
President    24000
Executive    12750
Manager       7835
Staff         3006
*/;

CREATE OR REPLACE VIEW EMPLOYEES_WITH_SALARIES AS
SELECT
    LAST_NAME, SALARY, SAL_CODE
FROM
    EMPLOYEES
INNER JOIN JOBS USING (JOB_ID)
INNER JOIN SALARIES USING (SAL_CODE);

DECLARE
    V_INCR INTEGER := 200;
BEGIN

for j in 1..2000 loop
    insert into employees (employee_id, last_name, original_salary, job_id)
    select employee_id + j*v_incr, last_name, original_salary, job_id
    from employees where employee_id < v_incr;
end loop;
end;
/
alter table jobs modify (job_id not null);
create unique index jobs_job_id on jobs(job_id);
alter table jobs
    add primary key (job_id)
    using index jobs_job_id;
alter table employees add (constraint employees_job_id_fk
    foreign key (job_id)
    references jobs(job_id));
alter table jobs modify (sal_code not null);
alter table salaries modify (sal_code not null);
create unique index salaries_sal_code on salaries(sal_code);
alter table salaries
    add primary key (sal_code)
    using index salaries_sal_code;
alter table jobs add (constraint jobs_sal_code_fk
    foreign key (sal_code)
    references salaries(sal_code));
select count(*) from employees /* 214107 */;
select Sum(salary) s from employees_with_salaries /* 1383403356 */;

IMPLEMENT THE TWO APPROACHES IN THE PAYROLL PACKAGE
alter session set plsql_compiler_flags = 'NATIVE';
create or replace package Payroll is
    procedure Time ( p_repeats in pls_integer );
    function By_Sql return pls_integer;
    function By_Plsql return pls_integer;
end Payroll;
/
create or replace package body Payroll is
t0 number; t1 number;
v_sum_salary number;
v_expected_sum constant number := 1383403356;
function By_Sql return pls_integer is
begin
  v_sum_salary := 0;
  t0 := Dbms_Utility.Get_Time();
  for j in ( select / *+use_hash(employees jobs salaries)*/ salary
    from
      employees
    inner join jobs using (job_id)
    inner join salaries using (sal_code) )
  loop
    v_sum_salary := v_sum_salary + j.salary;
  end loop;
  t1 := Dbms_Utility.Get_Time();
  if v_sum_salary <> v_expected_sum then
    Raise_Application_Error ( -20000,
      'Wrong v_sum_salary: ' || v_sum_salary ); end if;
  return ( t1 - t0 );
end By_Sql;

function By_Plsql return pls_integer is
  type sal_tab is table of salaries.salary%type
    index by employees.job_id%type;
  v_sal_tab sal_tab;
begin
  v_sum_salary := 0;
  t0 := Dbms_Utility.Get_Time();
  for j in ( select job_id, salary
    from jobs inner join salaries using (sal_code) )
  loop
    v_sal_tab(j.job_id) := j.salary;
  end loop;
  for j in ( select job_id from employees )
  loop
    v_sum_salary := v_sum_salary + v_sal_tab(j.job_id);
  end loop;
  t1 := Dbms_Utility.Get_Time();
  if v_sum_salary <> v_expected_sum then
    Raise_Application_Error ( -20000,
      'Wrong v_sum_salary: ' || v_sum_salary ); end if;
  return ( t1 - t0 );
end By_Plsql;

procedure Time ( p_repeats in pls_integer ) is
  type Time_Arr is table of pls_integer index by pls_integer;
  sql_times Time_Arr;
  plsql_times Time_Arr;
  avg_sql_time number := 0;
  avg_plsql_time number := 0;
begin
  sql_times(1)   := By_Sql;
  plsql_times(1) := By_Plsql; /* warm up */
  for j in 1..p_repeats loop
    sql_times(j)   := By_Sql;
    avg_sql_time   := avg_sql_time   + sql_times(j);
    plsql_times(j) := By_Plsql;
    avg_plsql_time := avg_plsql_time + plsql_times(j);
  end loop;
  Dbms_Output.Put_Line ( Lpad('By_Sql',10) ||
    Lpad('By_Plsql',10) );
for j in 1..p_repeats loop
    Dbms_Output.Put_Line ( Lpad(To_Char(sql_times(j), '99999'),10) ||
                          Lpad(To_Char(plsql_times(j),'99999'),10) );
end loop;

avg_sql_time  := Floor ( avg_sql_time / p_repeats );
avg_plsql_time := Floor ( avg_plsql_time / p_repeats );
Dbms_Output.Put_Line ( Lpad('----',10) ||
                          Lpad('----',10) );
Dbms_Output.Put_Line ( Lpad(To_Char(avg_sql_time, '99999'),10) ||
                          Lpad(To_Char(avg_plsql_time,'99999'),10) );
end Time;
end Payroll;
/
Set ServerOutput On Format Wrapped
call Payroll.Time(5);

A.2. CODE TO IMPLEMENT THE PAYROLL SCENARIO FROM SECTION 4

PERFECT TRIANGLES

Consider the following computationally intensive algorithm with no (requirement for) database access: find all right-angled triangles with all side lengths integer (a.k.a. a perfect triangle). We must count only unique triangles, i.e. those whose sides are not each the same integral multiple of the sides of a perfect triangle already found. The following implements an exhaustive search among candidate triangles with all possible combinations of lengths of the two shorter sides, each in the range 1 to a specified maximum. Each candidate is coarsely filtered by testing if the square root of the sum of the squares of the two short sides is within 0.01 of an integer. Triangles which pass this test are tested exactly by applying Pythagoras’s theorem using integer arithmetic. Candidate perfect triangles are tested against the list of multiples of perfect triangles found so far. Each new unique perfect triangle is stored in a PL/SQL table, and its multiples (up to the maximum length) are stored in a separate PL/SQL table to facilitate uniqueness testing.

The implementation thus involves a doubly nested loop with these steps at its heart: several arithmetic operations, casts and comparisons; calls to procedures implementing comparisons driven by iteration through a PL/SQL table (with yet more arithmetic operations); and extension of PL/SQL tables where appropriate.

procedure Perfect_Triangles ( p_max in pls_integer ) is
    -- Linear Scan in index-by-pls_integer table
    t1 pls_integer; t2 pls_integer;
    lng pls_integer; shrt pls_integer; hyp number; ihyp pls_integer;

type side_r is record ( shrt pls_integer, lng pls_integer );
type sides_t is table of side_r Index by pls_integer;
unique_sides sides_t;
m pls_integer:=0; -- curr max elements in "unique_sides"
dup_sides sides_t;
n pls_integer:=0; -- curr max elements in "dup_sides"

procedure Store_Dup_Sides  ( p_lng in pls_integer, p_shrt in pls_integer ) is
    mult pls_integer:=2;
    lng_mult pls_integer:=p_lng*2; shrt_mult pls_integer:=p_shrt*2;
begin
    while ( lng_mult < p_max ) or ( shrt_mult < p_max )
loop
        n := n+1;
        dup_sides(n).lng := lng_mult; dup_sides(n).shrt := shrt_mult;
        mult := mult+1; lng_mult := p_lng*mult; shrt_mult := p_shrt*mult;
    end loop;
end Store_Dup_Sides;

function Sides_Are_Unique ( p_lng in pls_integer, p_shrt in pls_integer ) is
return boolean is
begin
for j in 1..n
loop
  if ( p_lng = dup_sides(j).lng ) and ( p_shrt = dup_sides(j).shrt )
  then return false; end if;
end loop;
return true;
end Sides_Are_Unique;
end;

begin
  t1 := Dbms_Utility.Get_Time;
for lng in 1..p_max
loop
  for shrt in 1..lng
  loop
    hyp := Sqrt ( lng*lng + shrt*shrt ); ihyp := Floor(hyp);
    if hyp-ihyp < 0.01
    then
      if ( ihyp*ihyp = lng*lng + shrt*shrt )
      then
        if Sides_Are_Unique ( lng, shrt )
        then
          m := m+1;
          unique_sides(m).lng := lng; unique_sides(m).shrt := shrt;
          Store_Dup_Sides ( lng, shrt );
        end if;
      end if;
    end if;
  end loop;
end loop;
  t2 := Dbms_Utility.Get_Time;

  t1 := Dbms_Utility.Get_Time;
  for j in 1..m
  loop
    Dbms_Output.Put_Line ( Lpad(unique_sides(j).lng, 6,' ') ||
                          Lpad(unique_sides(j).shrt,6,' ') ) ;
  end loop;*/
  Dbms_Output.Put_Line ( 'No. of unique triangles:' || To_Char(m,99999) ||
                         chr(10) ||
                         'Elapsed time:           ' || To_Char( ((t2-t1)/100), '9999.9' ) || ' sec' );
end Perfect_Triangles;

--------------------------------------------------------------------------------
drop table dup_sides;
create global temporary table  dup_sides ( idx varchar2(4000) ) on commit delete rows;
create unique index dup_sides_idx on dup_sides(idx);

procedure Perfect_Triangles ( p_max in pls_integer ) is
  -- Using schema-level global temporary table with B*-tree index
  t1 pls_integer; t2 pls_integer;
lng pls_integer; shrt pls_integer; hyp number; ihyp pls_integer;

  type side_r is record ( shrt pls_integer, lng pls_integer );
type sides_t is table of side_r index by pls_integer;
unique_sides sides_t; m pls_integer:=0; -- curr max elements in "unique_sides"

v_idx dup_sides.idx%type;

procedure Store_Dup_Sides  ( p_lng in pls_integer, p_shrt in pls_integer ) is
  mult pls_integer:=2; lng_mult pls_integer:=p_lng*2; shrt_mult
  pls_integer:=p_shrt*2;
begin
  while ( lng_mult < p_max ) or ( shrt_mult < p_max )
    loop
      insert into dup_sides (idx) values ( lng_mult || '.' || shrt_mult );
      mult := mult+1; lng_mult := p_lng*mult; shrt_mult := p_shrt*mult;
    end loop;
  end Store_Dup_Sides;

function Sides_Are_Unique ( p_lng in pls_integer, p_shrt in pls_integer )
  return boolean is
begin
  begin
    select idx into v_idx from dup_sides where idx = p_lng || '.' || p_shrt;
  exception when no_data_found then
    return true;
  end;
  return false;
end Sides_Are_Unique;

begin
  execute immediate 'truncate table dup_sides';
t1 := Dbms_Utility.Get_Time;
for lng in 1..p_max
  loop
    for shrt in 1..lng
      loop
        hyp := Sqrt ( lng*lng + shrt*shrt ); ihyp := Floor(hyp);
        if hyp-ihyp < 0.01
          then
            if ( ihyp*ihyp = lng*lng + shrt*shrt )
              then
                if Sides_Are_Unique ( lng, shrt )
                  then
                    m := m+1;
                    unique_sides(m).lng := lng; unique_sides(m).shrt := shrt;
                    Store_Dup_Sides ( lng, shrt );
                  end if;
                end if;
              end if;
            end if;
          end loop;
        end loop;
t2 := Dbms_Utility.Get_Time;
/*for j in 1..m
  loop
    Dbms_Output.Put_Line ( 
      Lpad(unique_sides(j).lng, 6,' ') || 
      Lpad(unique_sides(j).shrt,6,' ') );
  end loop;*/
Dbms_Output.Put_Line ( 
  'No. of unique triangles:' || To_Char(m,99999) || 
  chr(10) ||

procedure Perfect_Triangles ( p_max in pls_integer ) is
-- Using index-by-varchar2 table
   t1 pls_integer; t2 pls_integer;
   lng pls_integer; shrt pls_integer; hyp number; ihyp pls_integer;

   type side_r is record ( shrt pls_integer, lng pls_integer );
   type sides_t is table of side_r index by pls_integer;
   unique_sides sides_t; m pls_integer:=0; -- curr max elements in "unique_sides"
   type dup_t is table of boolean index by varchar2(4000);
   dup_sides dup_t;

   procedure Store_Dup_Sides ( p_lng in pls_integer, p_shrt in pls_integer ) is
      mult pls_integer:=2; lng_mult pls_integer:=p_lng*2; shrt_mult
            pls_integer:=p_shrt*2;
      begin
         while ( lng_mult < p_max ) or ( shrt_mult < p_max )
            loop
               dup_sides ( lng_mult || '.' || shrt_mult ) := true;
               mult := mult+1; lng_mult := p_lng*mult; shrt_mult := p_shrt*mult;
            end loop;
      end Store_Dup_Sides;

   function Sides_Are_Unique ( p_lng in pls_integer, p_shrt in pls_integer )
            return boolean is
      begin
         return not dup_sides.Exists( p_lng || '.' || p_shrt );
      end Sides_Are_Unique;

      begin
         t1 := Dbms_Utility.Get_Time;
         for lng in 1..p_max
            loop
               for shrt in 1..lng
                  loop
                     hyp := Sqrt ( lng*lng + shrt*shrt ); ihyp := Floor(hyp);
                     if hyp-ihyp < 0.01
                        then
                           if ( ihyp*ihyp = lng*lng + shrt*shrt )
                              then
                                 if Sides_Are_Unique ( lng, shrt )
                                    then
                                       m := m+1;
                                       unique_sides(m).lng := lng;
                                       unique_sides(m).shrt := shrt;
                                       Store_Dup_Sides ( lng, shrt );
                                    end if;
                                 end if;
                           end if;
                     end if;
                  end loop;
         end loop;
         t2 := Dbms_Utility.Get_Time;
      end loop;

      /*for j in 1..m
         loop
            Dbms_Output.Put_Line ( Lpad(unique_sides(j).lng, 6, ' ') ||
            Dbms_Output.Put_Line ( Lpad(unique_sides(j).shrt, 6, ' ') ||

Lpad(unique_sides(j).shrt,6,' ') );
end loop;/*
Dbms_Output.Put_Line ( 'No. of unique triangles: ' || To_Char(m,99999) ||
chr(10) ||
'Elapsed time: ' || To_Char( ((t2-t1)/100), '9999.9' ) || ' sec' );
end Perfect_Triangles;