EFFECTIVE UTILIZATION OF THE DATABASE IN WEB DEVELOPMENT

Dr. Paul Dorsey, Dulcian, Inc.
Michael Rosenblum, Dulcian, Inc.

The Fusion technology stack is large and complex. It is often hard for many IT professionals to make the transition into the J2EE environment, given the host of different tools, programming languages, architectures, and technologies. To make the situation even more difficult, many application development projects often have the illusion of progress even though they may be on the path to ultimate failure. With all of these potential pitfalls, building functioning, scalable production software often becomes an impossible task.

Oracle provides a first-rate Service Oriented Architecture (SOA)-centric environment, built from an Object Oriented (OO) developer's perspective. Therefore, it lacks much of the vision that would make experienced Oracle Designer users comfortable. There is also a "not-so-subtle" encouragement to place business rules enforcement in the middle tier, mostly implemented as Java-code. The SOA approach can be used to articulate data-centric complex business processes, using portions of the architecture such as Oracle's Business Process Execution Language (BPEL). However, this can lead to the development of applications with poor performance because of the number of round trips needed between the middle tier and the database. This paper describes a "Thick Database" or "Micro-Service-Oriented-Architecture" (M-SOA) approach to application development which can be used to overcome many of the performance issues and provide a robust architecture for building systems with Fusion Middleware.

"Thick Database" Thinking

Using a thick database approach, there is a clear division between the database and user interface (UI) portions of an application. The two key features of this approach are as follows:

1. Nothing in the UI ever directly interacts with a database table. All interaction is accomplished through database views or APIs.
2. Nearly all application behavior (including screen navigation) is handled in the database.

Separating the database and UI portions of the development process makes sense, since the likelihood of changing databases is much less than that of changing UI platforms. Over time, although Oracle will add features, the DBMS will not internally refactor. The existing stack "works." Changing databases implies a huge DBA learning curve and very high costs. In contrast, the UI technology stack is in a constant state of flux. Will you need to use Java EE or .Net? What about APEX or FLEX? Since all of the UI environments change, many system redesigns are assured.

By separating the user interface and the database logic, the result is a simpler architecture. Ostensibly, you are building two smaller applications, which is simpler than building one large one. The UI can then be built more easily and quickly and can be shown to users right away. This provides faster feedback to the development team and helps to identify design errors much earlier in the process

Many organizations where application development architecture is driven by Java or .Net programmers have been lured into ignoring the database. For large systems there is no logical argument for database independence.

However, do not be fooled into thinking that using a thick database approach means simply stuffing everything into the database and hoping for the best. The system must be carefully architected and implemented in order to take advantage of all of the benefits of this approach.

Benefits of the Thick Database Approach

Using a thick database approach has many advantages:

1. It makes your application UI technology-independent.
2. It creates reusable, UI technology-independent views and APIs.

3. It reduces the complexity of UI development.
4. The database provides needed objects.
5. Using this approach reduces the burden on the UI developer.
6. This approach minimizes project development risk.
7. This approach helps build working applications that scale well with:
   - Better performance (10X)
   - Less network traffic (100X)
   - Less code (2X)
   - Fewer application servers (3X)
   - Fewer database resources (2X)
   - Faster development (2X)

Less PL/SQL code is needed in order to perform data-centric operations than is the case in Java. There are also more “tricks” available to make the PL/SQL code run faster. Database-intensive code can always be written more efficiently in the database. Less code means less coding time.

Another huge benefit of using the thick database approach is that the system will be much easier to refactor since UI technology stack changes are common and the .Net/Java EE battle rages on. Since Web architecture is much more volatile than the database platform, using a thick database approach is a good defense against the chaos of a rapidly evolving standard.

How Thick is Too Thick?

What would happen if 100% of all UI logic were placed in the database? This would be a pathologically complete way to implement the thick database approach. For example, the following front-end activities would need to be resolved in the database every time, requiring additional round-trips:

- Tabbing out of a field
- Population of an LOV
- Page navigation

A system built this way would be sub-optimal for obvious reasons. However, it is possible to create a working system using this strategy.

How Thin is Too Thin?

Successfully building applications that are 100% database “thin” requires a highly skilled team. It is necessary to write the code very efficiently in order to minimize database round trips. Any middle tier technology (e.g. BPEL) can also be a performance killer. This approach is possible but difficult to implement.

The Thick Database Development Process

Using a thick database approach, the two portions of an application can be coded independently. The UI and database development take place at the same time. Teams can work in isolation until substantive portions are working. This way, the first version of the UI is built within a few days. This UI can also be used as testing environment for the database team. Another advantage of the speedy development time is that feedback can be quickly received from users.

Using an Agile development process works well with thick database development. Minimal design work can be done rapidly to produce a partially working system. Additional functionality is created using an iterative design process. The interface code can be stubbed out for the views and APIs using the following code models:

```
Create or replace view <…> as
select api_pkg.f_getValue1 val1
       api_pkg.f_getValue2 val2
       …
from dual
```

APIs are functions that return a correct value (usually hard-coded). Interfaces will change as the application matures.

The UI team takes the APIs and incorporates them into the application. Then, the database team makes them work.

Using Denormalized Views

The idea here is to convert relational data into something that will make user interface development easier. This is the easiest way to separate the data representation in the front-end from the real model. The solution is to specify all columns from all tables (with appropriate joins) that would be needed and create a view as shown here:

```
create or replace view v_customer
as
select c.cust_id,
       c.name_tx,
       a.addr_id,
       a.street_tx,
       a.state_cd,
       a.postal_cd
from customer c
left outer join address a
   on c.cust_id = a.cust_id
```

To make this view updatable, a special kind of trigger (INSTEAD-OF) should be created for that view.

Using INSTEAD OF Inserts

The INSTEAD OF code construction can be used to Insert, Update and Delete as shown in the following three code samples:

INSTEAD OF - Insert

```
create or replace trigger v_customer_ii
instead of insert on v_customer
declare
  v_cust_id customer.cust_id%rowtype;
begin
  if :new.name_tx is not null then
   insert into customer (cust_id,name_tx)
    values(object_seq.nextval,            :new.name_tx)
   returning cust_id into v_cust_id;
  if :new.street_tx is not null then
   insert into address (addr_id,street_tx,
        state_cd, postal_cd, cust_id)
   values (object_seq.nextval,:new.street_tx,
    :new.state_cd,:new.postal_cd, v_cust_id);
  end if;
end;
```

INSTEAD OF - Delete

```
create or replace trigger v_customer_id
instead of delete on v_customer
begin
    delete from address
    where cust_id=:old.cust_id;
    delete from customer
    where cust_id=:old.cust_id;
end;
```

INSTEAD OF - Update

```
create or replace trigger v_customer_iu
instead of update on v_customer
```

```
begin
 update customer set name_tx  = :new.name_tx
 where cust_id = :old.cust_id;

 if :old.addr_id is not null
 and :new.street_tx is null then
  delete from address where addr_id=:old.addr_id;
 elsif :old.addr_id is null
 and :new.street_tx is not null then
  insert into address(…) values (…);
 else
  update address set… where addr_id=:old.addr_id;
 end if;
end;
```

Function-Based Views

Sometimes it is just not possible to represent all required functionality in a single SQL statement, which means that a denormalized view cannot be built. But Oracle provided a different mechanism, namely collections, that not only allow you to hide the data separation, but also all of the transformation logic.

Using Collections

A collection is an ordered group of elements, all of the same type, addressed by a unique subscript consisting of arrays of primitives (VARCHAR2, date….) and records. Collections allow you to articulate and manipulate sets of data. Since all collections represent data, they are defined as data types. The advantages of using collections are that they are usually faster, create cleaner code, and are well suited for UI views. Processing data in sets is “usually” faster than doing so one element at a time. Manipulating sets in memory is “usually” 100 times faster than manipulating sets on the storage device.

Unfortunately, collections are not always faster and require using a somewhat annoying syntax. The amount of memory is limited (especially in 32-bit architecture). Although storage is cheap, memory is not. Also, using collections can have a steep learning curve. People who are used to processing one row at a time (since COBOL days) may have problems working with sets.

There are three types of collections: nested tables, associative arrays, and variable-size arrays (v-arrays). The last type is rarely used outside of mathematical tasks, so this paper will only discuss the first two.

Using Nested Tables and Corresponding Functions

Nested tables consist of arbitrary groups of elements of the same type with sequential numbers as a subscript. An undefined number of elements can be added or removed on the fly. Nested tables are available in SQL and PL/SQL. They are very useful in PL/SQL, but not in tables. An example of code for using nested tables is as follows:

```
declare
  type NestedTable is table of ElementType;
...
create or replace type NestedTable is table of ElementType;
```

Nested tables are NOT dense. You can remove objects from inside of the array and the size of the nested table may or may not equal the subscript of the last element. You can use a built-in NEXT and PREVIOUS command to cover this gap as shown here:

```
declare
    type month_nt is table of VARCHAR2(20);
    v_month_nt month_nt:=month_nt();
    i number;
begin
    v_month_nt.extend(3);
    v_month_nt(1):='January';
    v_month_nt(2):='February';
```

```
      v_month_nt(3):='March';
      v_month_nt.delete(2);
      DBMS_OUTPUT.put_line('Count:'||v_month_nt.count);
      DBMS_OUTPUT.put_line('Last:'||v_month_nt.last);
      i:=v_month_nt.first;
      loop
          DBMS_OUTPUT.put_line(v_month_nt(i));
          i:=v_month_nt.next(i);
          if i is null then exit;
          end if;
      end loop;
  end;
```

Nested tables can be used in SQL queries with the special operator TABLE. This allows you to hide the complex procedural logic "under the hood." The nested table type must be declared as a user-defined type (CREATE OR REPLACE TYPE…) You need to specify exactly what is needed as output and declare the corresponding collection as shown here:

```
Create type lov_oty is object (id_nr NUMBER, display_tx VARCHAR2(256));

Create type lov_nt as table of lov_oty;
```

To write a PL/SQL function that hides all required logic, you can use code like the example shown here:

```
function f_getLov_nt (i_table_tx,i_id_tx,i_display_tx,i_order_tx)
return lov_nt is
  v_out_nt lov_nt := lov_nt();
begin
  execute immediate
    'select lov_oty('||i_id_tx||','||i_display_tx||')'||
    ' from '||i_table_tx||
    ' order by '||i_order_tx
  bulk collect into v_out_nt;
  return v_out_nt;
end;
```

Now there is a function that can return an object collection. An object collection can be cast to a table. Object collection types are supported in SQL. To summarize all three statements: a function call could be used as a data source in the SQL. You can test it using the following code:

```
select id_nr, display_tx
from table(
           cast(f_getLov_nt
                  ('emp',
                   'empno',
                   'ename||''-''||job',
                   'ename')
           as lov_nt)
           )
```

Creating a View

Finally, create a view on top of the SQL statement. This completely hides the underlying logic from the user interface. INSTEAD-OF triggers make the logic bi-directional. One minor problem is that there is still no way of passing parameters into the view other than by using some kind of global as shown here:

```
Create or replace view v_generic_lov as
select id_nr, display_tx
from table( cast(f_getLov_nt
                       (GV_pkg.f_getCurTable,
                        GV_pkg.f_getPK(GV_pkg.f_getCurTable),
                        GV_pkg.f_getDSP(GV_pkg.f_getCurTable),
                        GV_pkg.f_getSORT(GV_pkg.f_getCurTable)
                       )
                as lov_nt
                    )
          )
```

Optimization of Database Processing

There are some techniques discussed in this section that may not be translated into direct SQL statements, but as part of PL/SQL logic behind the scenes, they can be extremely useful (especially performance-wise).

Using Associative Arrays

An associative array is a collection of elements that uses arbitrary numbers and strings for subscript values. Associative arrays can only be used in PL/SQL and are still useful. A code example is shown here:

```
declare
  type NestedTable is table of ElementType index by Varchar2([N]);
...
 type NestedTable is table of ElementType index by binary_integer;


declare
  type dept_rty is record (deptNo number, extra_tx VARCHAR2(2000));
  type dept_aa is table of dept_rty index by binary_integer;
  v_dept_aa dept_aa;
begin
  for r_d in (select deptno from dept)  loop
    v_dept_aa(r_d.deptno).deptNo:=r_d.deptno;
  end loop;

  for r_emp in (select ename, deptno from emp) loop
    v_dept_aa(r_emp.deptNo).extra_tx:=
        v_dept_aa(r_emp.deptNo).extra_tx||' '||r_emp.eName;
  end loop;
end;
```

You need to index by VARCHAR2 instead of by BINARY_INTEGER. Associative arrays cannot be used in a FOR-loop. You must also allow for the creation of simple composite keys with direct access to the row in memory. In order to prepare the memory structure, use the following code:

```
declare
  type list_aa is table of VARCHAR2(2000) index by VARCHAR2(256);
  v_list_aa list_aa;
  cursor c_emp is
  select ename, deptno,to_char(hiredate,'q') q_nr
  from emp;
  v_key_tx VARCHAR2(256);
begin
  for r_d in (select deptno from dept order by 1) loop
    v_list_aa(r_d.deptno||'|1'):='Q1 Dept#' ||r_d.deptno||':';
    v_list_aa(r_d.deptno||'|2'):='Q2 Dept#' ||r_d.deptno||':';
    v_list_aa(r_d.deptno||'|3'):='Q3 Dept#' ||r_d.deptno||':';
    v_list_aa(r_d.deptno||'|4'):='Q4 Dept#' ||r_d.deptno||':';
  end loop;
```

To process data and present the results, you can use code like the following:

```
  ...
 for r_emp in c_emp loop
   v_list_aa(r_emp.deptno||'|'||r_emp.q_nr):=
      list_aa(r_emp.deptno||'|'||r_emp.q_nr)||' '||r_emp.ename;
 end loop;
 v_key_tx:=v_list_aa.first;
 loop
```

```
      DBMS_OUTPUT.put_line(v_list_aa(v_key_tx));
      v_key_tx:=v_list_aa.next(v_key_tx);
      exit when v_key_tx is null;
    end loop;
   end;
```

Using Bulk Operations

Bulk operations can be performed on sets using the following process:

- BULK loading into the memory
- BULK processing
- Manipulation of SETs (MULTISET operators)

Using the BULK COLLECT clause fetches a group of rows all at once to the collection. You can also control the number of fetched rows using the LIMIT command. The risks are that using BULK COLLECT does not raise the NO_DATA_FOUND message and the machine can run out of memory.

The syntax for using BULK COLLECT is as follows:

```
select …
bulk collect into Collection
 from Table;

update …
returning … bulk collect into Collection;

fetch Cursor
bulk collect into Collection;

declare
  type emp_nt is table of emp%rowtype;
  v_emp_nt emp_nt;

  cursor c_emp is select * from emp;
begin
  open c_emp;
  loop
    fetch c_emp
    bulk collect into v_emp_nt limit 100;
    p_proccess_row (v_emp_nt);
    exit when c_emp%NOTFOUND;
  end loop;
  close c_emp;
end;
```

Using the FORALL command

The idea here is to apply the same action for all elements in the collection. You should have only one context switch between SQL and PL/SQL. There are some risks associated with using this command. Special care is required if only some actions from the set succeeded. The syntax is shown here:

```
 forall Index in lower..upper
  update … set … where id = Collection(i)
...
 forall Index in lower..upper
  execute immediate '…' using Collection(i);
```

There are also some important restrictions to keep in mind when using the FORALL command:

- Only a single command can be executed.
- You must reference at least one collection inside the loop.

- All subscripts between lower and upper limits must exist.
- This does not work with the associative array INDEX BY VARCHAR2
- You cannot use the same collection with SET and WHERE.
- You cannot refer to the individual column on the object/record (only the whole object)

An example of the FORALL syntax is as follows:

```
declare
    type number_nt is table of NUMBER;
    v_deptNo_nt number_nt:=number_nt(10,20);
begin
    forall i in v_deptNo_nt.first()..v_deptNo_nt.last()
      update emp
        set sal=sal+10
      where deptNo=v_deptNo_nt(i);
end;
```

MULTISET operations

The idea behind using MULTISET operations is to combine the results of two nested tables into a single nested table. The available commands to do this are as follows:

- MULTISET EXCEPT
- MULTISET INTERSECT
- MULTISET UNION

An example of the syntax for using MULTISET is as follows:

```
declare
    v_emp1_nt number_nt;
    v_emp2_nt number_nt;
    v_emp3_nt number_nt;
begin
    select empno bulk collect into v_emp1_nt
    from emp where sal>1000;
    select empno bulk collect into v_emp2_nt
    from emp where job!='MANAGER';
    select v_emp1_nt multiset union distinct v_emp2_nt
    into v_emp3_nt from dual;
dbms_output.put_line('A OR B: '||v_emp3_nt.count);
    select v_emp1_nt multiset except distinct v_emp2_nt
    into v_emp4_nt from dual;
dbms_output.put_line('A MINUS B: '||v_emp3_nt.count);
    select v_emp1_nt multiset intersect distinctv_emp2_nt
    into v_emp5_nt from dual;
dbms_output.put_line('A AND B: '||v_emp3_nt.count);
end;
```

Conclusions

It seems logical that larger systems would use a hybrid approach, if not a purely thick database approach. Unfortunately, this is often not the case. Java and .Net programmers tend to think that everything should be written in either Java or .Net. Therefore, in both environments because of ignorance of database capabilities and a herd mentality, these IT professionals tend towards database-independence. This attitude has become so pervasive that the database is often viewed as irrelevant and good database design is often ignored when building systems.

The #1 critical success factor for any Web development is effective utilization of the database. Just because everyone is moving logic to the middle tier does not make it a smart idea. PL/SQL efficiency continues to improve. Code that needs to access the database runs much faster if it is located the database.

Database independence is irrelevant. The likelihood of an organization changing its database platform is much less than that of changing its UI technology. Therefore, building systems that are UI-independent is a much more logical strategy to pursue in order to avoid frequent system rewrites.

About the Authors

Dr. Paul Dorsey is the founder and president of Dulcian, Inc., an Oracle consulting firm specializing in business rules and Web-based application development. He is the chief architect of Dulcian's Business Rules Information Manager (BRIM®) tool. Paul is the co-author of seven Oracle Press books on Designer, Database Design, Developer, and JDeveloper, which have been translated into nine languages as well as the Wiley Press book *PL/SQL for Dummies*. Paul is an Oracle ACE Director. He is President Emeritus of NYOUG and the Associate Editor of the International Oracle User Group's SELECT Journal. Paul was honored by ODTUG as volunteer of the year in 2003 and as Best Speaker (Topic & Content) for the 2007 conference, in 2001 by IOUG as volunteer of the year and by Oracle as one of the six initial honorary Oracle 9*i* Certified Masters. Paul is also the founder and Chairperson of the ODTUG Symposium, currently in its ninth year. Paul's submission of a Survey Generator built to collect data for The Preeclampsia Foundation was the winner of the 2007 Oracle Fusion Middleware Developer Challenge and Oracle selected him as the 2007 PL/SQL Developer of the Year.

Michael Rosenblum is a Development DBA at Dulcian, Inc. He is responsible for system tuning and application architecture. He supports Dulcian developers by writing complex PL/SQL routines and researching new features. Michael is the co-author of *PL/SQL for Dummies* (Wiley Press, 2006). Michael is a frequent presenter at various regional and national Oracle user group conferences. In his native Ukraine, he received the scholarship of the President of Ukraine, a Masters Degree in Information Systems, and a Diploma with Honors from the Kiev National University of Economics.