Customised Object Monitoring

As a Systems Implementer I often have to help my clients with a difficult decision: should we customise some Oracle-supplied code? "No!" I hear you cry, "You can't do that! Oracle won't support you!" Well, sometimes to achieve a certain piece of functionality customers are prepared to accept the risks that come with customisation.

Barry Goodsell,
Technical Manager,
Projected Consulting Ltd

One of the biggest problems to overcome is dealing with the fact it is almost guaranteed that your customised code will be overwritten by an Oracle patch at some point in the future. It is often down to the Developer or Functional Consultant to look at the Release Notes for a patch to try and determine whether a certain customisation might be affected. For small patches this isn't really a problem, but with the larger rollup patches the impact analysis becomes quite a major task. But even then, that approach can be problematic – sometimes Oracle, in their infinite wisdom, release new versions of code objects that are not even mentioned in the Release Notes. The upshot of this is that the only sure way to find out what is impacted is to search through every single file included in the patch – a lengthy and tedious exercise.

I have recently been working with a client who has quite a large number of customisations that have been in place since "go-live" eighteen months ago. The system is now stable after the initial implementation issues and the organisation have started to try and catch-up with the patches that they had neglected to apply whilst trying to get to "business as usual". Performing the impact analyses of the patches was taking up a considerable amount of time and so I proposed an automated solution, which I have been kindly given permission to share with you.

The Problem

The main problem is that we need to capture the "state" of a customised object just after it has been installed, so that we can compare that state with the state after a patch has been applied. If the states are different then we know that Oracle has overwritten our customisation. My first version of this utility used the LAST_DDL_TIME from ALL_OBJECTS. However, this was not entirely reliable as this timestamp is changed when an object is compiled, even if it hasn't changed.

It was when I was browsing through the PL/SQL Package and Types Manual for 10g Database when I came across the new DBMS_CRYPTO package. DBMS_CPYTPO (and it's quaintly named predecessor DBMS_OBFUSCATION_TOOLKIT) is used to encrypt and decrypt data, using a variety of industry standard encryption and hashing algorithms. You may need to ask your DBA to grant you EXECUTE rights on it, as by default it is not exposed to all users.

DBMS_CRYPTO contains a HASH function which can create a hash of a RAW, BLOB or a CLOB returning a RAW. The function supports three different hashing algorithms: MD4, MD5 and SHA1. The two Message Digest (MD) algorithms return a 128-bit hash value; the Secure Hash Algorithm returns a 160-bit hash value. When investigating whether to use hashes, I was concerned that two different bits of code could potentially

Hash Function

A hash function is any well-defined procedure or mathematical function for turning some kind of data into a relatively small integer, which may serve as an index into an array. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

Hash functions are mostly used to speed up table lookup or data comparison tasks — such as finding items in a database, detecting duplicated or similar records in a large file, finding similar stretches in DNA sequences, and so on.

Extract courtesy of wikipedia.org

produce the same hash value. To see how many different hash values can be generated from a function we need to look at the size of the returned hash values.

For MD4/5: $2^{128} = 3.4 \times 10^{38}$
For SHA: $2^{160} = 1.5 \times 10^{48}$

Both of these are very large numbers indeed! You stand more chance of winning the lottery jackpot two weeks in a row than ever hitting the same hash value. I decided to go with SHA because 160-bits is only 20-bytes, not a lot of storage required for each object that we want to monitor.

The Solution

Now to write some code! The first function that I wrote (GET_HASH) just encapsulated the call to the HASH function taking in a CLOB parameter. I decided to use the CLOB variant as I thought that it would be easiest to create CLOB representations of the objects that I want to monitor.

Still using 9i?

DBMS_CRYPTO is new to Oracle10g which doesn't help those of you still on Oracle9i.

You could use DBMS_UTILITY.GET_HASH_VALUE, but this only accepts VARCHAR2 parameters.

In the past I have used a Java Stored Procedure to hash a VARCHAR2 string; without too much work it could be converted to process a CLOB.

The next problem was how to how to actually get these representations out of the database. A lot of the things that I wanted to monitor are found in the Data Dictionary, but some things like Workflows and Alerts, are stored in E-Business Suite tables where there are quite complex relationships between the tables.

Having been working with XML Publisher quite extensively over the last few years I thought that XML would be the right way to go. As it happens, Oracle Server provides quite a number of different ways of creating XML from the database – the package that I was most interested in was DBMS_XMLGEN. This package, to quote the manual, "converts the results of a SQL query into a canonical XML format. The package takes an arbitrary SQL query as input, converts it to XML format, and returns the result as a CLOB" – exactly what I needed:

```
FUNCTION get_xml
  ( p_sql          IN           VARCHAR2 )
RETURN CLOB
IS
  l_ctx     dbms_xmlgen.ctxhandle;
  l_result  CLOB;

BEGIN
  l_ctx := dbms_xmlgen.newContext(p_sql);
  dbms_xmlgen.setConvertSpecialChars(l_ctx, TRUE);
  l_result := dbms_xmlgen.getXML(l_ctx);

  dbms_xmlgen.closeContext(l_ctx);

  RETURN l_result;
END get_xml;
```

I then needed a function to build a suitable SQL statement given a table name and one or more primary key column names and values. This could then call GET_XML and GET_HASH and return the hash value:

```
PROCEDURE add_condition
  ( p_sql                   IN OUT NOCOPY VARCHAR2
  , p_pk_column             IN            VARCHAR2
  , p_pk_value              IN            VARCHAR2 )
IS
BEGIN
  IF (p_pk_column IS NOT NULL)
  THEN
    p_sql := p_sql || ' AND '||p_pk_column||' = ''''||p_pk_value||'''';
  END IF;
END add_condition;

FUNCTION calculate_data_hash
  ( p_table_name            IN            VARCHAR2
  , p_pk_column1            IN            VARCHAR2  DEFAULT NULL
  , p_pk_value1             IN            VARCHAR2  DEFAULT NULL
  , p_pk_column2            IN            VARCHAR2  DEFAULT NULL
  , p_pk_value2             IN            VARCHAR2  DEFAULT NULL
  , p_pk_column3            IN            VARCHAR2  DEFAULT NULL
  , p_pk_value3             IN            VARCHAR2  DEFAULT NULL )
RETURN RAW
IS
  l_sql     VARCHAR2(2000);

BEGIN
  l_sql := 'SELECT * FROM '||p_table_name||' WHERE 1=1';

  add_condition(l_sql, p_pk_column1, p_pk_value1);
  add_condition(l_sql, p_pk_column2, p_pk_value2);
  add_condition(l_sql, p_pk_column3, p_pk_value3);

  RETURN get_hash(get_xml(l_sql));

END calculate_data_hash;
```

Using the above CALCULATE_DATA_HASH function we can calculate the hash value for almost any type of database object. We just need to pass in the database table or view name and enough column names and values to uniquely indentify our object:

```
FUNCTION calculate_hash
  ( p_object_type           IN        VARCHAR2
  , p_object_name           IN        VARCHAR2
  , p_table_name            IN        VARCHAR2  DEFAULT NULL
  , p_pk_column1            IN        VARCHAR2  DEFAULT NULL
  , p_pk_value1             IN        VARCHAR2  DEFAULT NULL
  , p_pk_column2            IN        VARCHAR2  DEFAULT NULL
  , p_pk_value2             IN        VARCHAR2  DEFAULT NULL
  , p_pk_column3            IN        VARCHAR2  DEFAULT NULL
  , p_pk_value3             IN        VARCHAR2  DEFAULT NULL )
RETURN RAW
IS
  l_hash_value  RAW(40) := NULL;

BEGIN
  CASE
    WHEN (p_object_type = 'DATA')
    THEN
      l_hash_value := calculate_data_hash
        ( p_table_name  => p_table_name
        , p_pk_column1  => p_pk_column1
        , p_pk_value1   => p_pk_value1
        , p_pk_column2  => p_pk_column2
        , p_pk_value2   => p_pk_value2
        , p_pk_column3  => p_pk_column3
        , p_pk_value3   => p_pk_value3 );

    WHEN (p_object_type IN
      ('PROCEDURE','FUNCTION','PACKAGE','PACKAGE BODY','TYPE','TYPE
BODY'))
    THEN
      l_hash_value := calculate_data_hash
        ( p_table_name  => 'ALL_SOURCE'
```

>>

```
        , p_pk_column1  => 'TYPE'
        , p_pk_value1   => p_object_type
        , p_pk_column2  => 'NAME'
        , p_pk_value2   => p_object_name );

    WHEN (p_object_type = 'VIEW')
    THEN
      l_hash_value := calculate_data_hash
        ( p_table_name  => 'ALL_VIEWS'
        , p_pk_column1  => 'VIEW_NAME'
        , p_pk_value1   => p_object_name );

    WHEN (p_object_type = 'MATERIALIZED VIEW')
    THEN
      l_hash_value := calculate_data_hash
        ( p_table_name  => 'ALL_MVIEWS'
        , p_pk_column1  => 'MVIEW_NAME'
        , p_pk_value1   => p_object_name );

    WHEN (p_object_type = 'TRIGGER')
    THEN
      l_hash_value := calculate_data_hash
        ( p_table_name  => 'ALL_TRIGGERS'
        , p_pk_column1  => 'TRIGGER_NAME'
        , p_pk_value1   => p_object_name );

    WHEN (p_object_type = 'MESSAGE')
    THEN
      l_hash_value := calculate_data_hash
        ( p_table_name  => 'FND_NEW_MESSAGES'
        , p_pk_column1  => 'MESSAGE_NAME'
        , p_pk_value1   => p_object_name );

    ELSE
      l_hash_value := NULL;

  END CASE;

  RETURN l_hash_value;

END calculate_hash;
```

Now that we can create the hash value for a wide variety of objects, we need somewhere to store those values. I created a table called XXC_CUSTOMISED_OBJECTS which had columns for the Object Name and Type, the Table Name and three pairs of Primary Key Column Names and values. To make life easier for the developers, I also created a simple API (REGISTER_OBJECT) to either create or update a hash record in this table. This API is to be called immediately after any customised object has been created, to store the hash of the newly created object in the table.

We now have a table holding all the information about the objects that we want to monitor and their current hash values. After an Oracle-supplied patch has been applied to our Patch Test environment, we can re-calculate the hash values to see whether anything has changed. To make life simpler, I added a function that returns the current (as opposed to the stored) hash value for a given CUSTOM_OBJECT_ID.

```
FUNCTION calculate_hash
  ( p_custom_object_id      IN        NUMBER )
RETURN RAW
IS
  r_object  xxc_customised_objects%ROWTYPE;

BEGIN
  SELECT  cob.*
  INTO    r_object
  FROM    xxc_customised_objects cob
  WHERE   cob.custom_object_id   = p_custom_object_id;

  RETURN calculate_hash
    ( p_object_type => r_object.object_type
    , p_object_name => r_object.object_name
    , p_table_name  => r_object.table_name
    , p_pk_column1  => r_object.pk_column1
    , p_pk_value1   => r_object.pk_value1
    , p_pk_column2  => r_object.pk_column2
    , p_pk_value2   => r_object.pk_value2
    , p_pk_column3  => r_object.pk_column3
    , p_pk_value3   => r_object.pk_value3 );

END calculate_hash;
```

To determine whether any objects have changed, I wrote the following view which calls the above function; any records where the STORED_HASH_VALUE is different from CURRENT_HASH_VALUE point to objects that have been altered since creation:

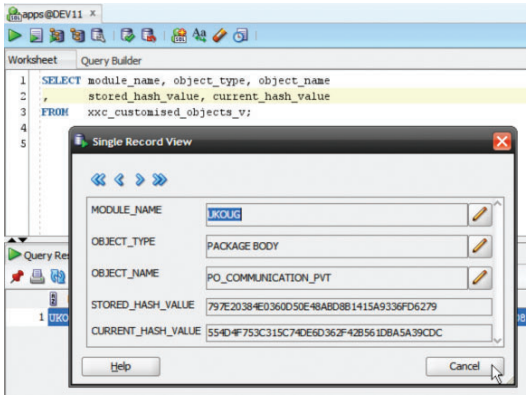
```
CREATE OR REPLACE VIEW xxc_customised_objects_v
AS
SELECT  OBJ.*
,       DECODE(OBJ.stored_hash_value
          , OBJ.current_hash_value, 'N'
          , 'Y') AS object_changed
FROM (
  SELECT  cob.module_name
  ,       cob.object_type
  ,       cob.object_name
  ,       cob.table_name
  ,       cob.pk_column1
  ,       cob.pk_value1
  ,       cob.pk_column2
  ,       cob.pk_value2
  ,       cob.pk_column3
  ,       cob.pk_value3
  ,       cob.hash_value                AS stored_hash_value
  ,       xxc_custom.calculate_hash
            (cob.custom_object_id)      AS current_hash_value
  FROM    xxc_customised_objects cob
) OBJ
ORDER BY OBJ.module_name, OBJ.object_type, OBJ.object_name
```

Example

Let's assume that you've made a customisation to the package body of PO_COMMUNICATION_PVT to enable extra attributes on the Purchase Order Print. After you've installed your customised version you would need to create and store a hash of the current package body:

```
BEGIN
  xxc_custom.register_object
    ( p_module_name => 'UKOUG'
    , p_object_type => 'PACKAGE BODY'
    , p_object_name => 'PO_COMMUNICATION_PVT' );
END;
```

At some point in the future, you have applied an Oracle patch and you want to check to see if the customised package body has been over-written: just select from the view and you can see that the hashes don't match, so you will need to re-apply your customisations to the new version:



Conclusion

This is a relatively simple, non-invasive method for monitoring changes to database objects. It has the potential to eliminate the many hours normally spent examining the contents of Oracle patches by quickly providing you with a list of objects that have been changed.

The process is simple to extend to monitor other object types by adding further sections to the first overload of the CALCULATE_HASH function. In fact, the version that you can download from the UKOUG website has support for Workflows and Alerts. This takes advantage of the fact that the DBMS_XMLGEN package fully supports nested cursor statements in the SQL, which can give you an XML representation of a complete relational data structure. ■

ABOUT THE AUTHOR

Barry Goodsell
Technical Manager, Projected Consulting

Barry Goodsell has over 20 years of experience with the Oracle database and development tools. He has been developing and supporting extensions to E-Business Suite for over 15 years, with his experience ranging from Release 10.7SC through to Release 12. He is an expert in all of Oracle's development tools and languages, including SQL, PL/SQL, Oracle Forms and Reports, Workflow, Oracle Discoverer, Java and JDeveloper. Barry specialises in developing web user interface extensions to E-Business Suite using Oracle Application Framework (OAF) and reporting solutions using OBIEE and BI Publisher.