Managing Database Change With Oracle Change Manager

© Geoff Ingram, Proton Technology Ltd 2002. No part of this document may be reproduced for any purpose without the express written permission of Proton Technology Limited.

All applications evolve over time, as bugs are fixed and enhancements are made. The application may be developed in house or by a third party. The changes may involve database schema changes or changes to data contents. It is critical that you manage change in such a way that undesired side effects and potential outages can be avoided.

Although the change management process is critical to minimizing outages, too often it’s viewed as something that simply obstructs "real work." This paper covers the requirements and attributes of the change management process, as well as how to manage Oracle database management system (DBMS) changes, including tools and techniques for doing so. These tools and techniques range from simple Structured Query Language (SQL)-based techniques, used to compare production and development versions of schemas, right up to Oracle Change Manager. Oracle Change Manager is an industrial-strength graphical user interface (GUI) tool (part of Oracle Enterprise Manager [OEM]) that provides everything you need to manage Oracle schema changes and versioning. Although you can skip straight to the section on Oracle Change Manager, it's useful to have a broad understanding of the requirements for a change management process first, and be aware of SQL-based techniques for carrying a quick basic schema comparison, as well as the extensive object reverse engineering capabilities of the DBMS_META package in Oracle9i. The list of topics covered is as follows:

- The change management overview
- The change management process
- Understanding object dependencies
Comparing object definitions and metadata using SQL

Reverse engineering object definitions using DBMS_META in Oracle9i

Using Oracle Change Manager based on a step-by-step example

**Change Management Overview**

The change management process is business critical, although it is often perceived as cumbersome. It's important that you quantify the potential risk of change. Quantifying change risk is difficult, and putting contingency plans in place involves hard work. Too often, those requirements are left out, sometimes with disastrous results. This section sets out the main requirements for a change management process.

**Tools for Change Management**

From my experience, one of the main reasons for resistance to change management is the lack of suitable tools for managing the process. Any software tool that is used enterprise-wide needs to be very easy to use. If it takes a database administrator (DBA) two hours to create a simple change, then there is definitely something wrong with the tools, or process, or both. Overly complex and difficult-to-use user interfaces are one of the main reasons for resistance to change management. If technical people respond with a sigh when change management is required, then something is wrong. The creation of a change management tool requires a professional user interface design and lots of usability testing, which are both expensive and time consuming. But the repercussions of choosing technology that doesn't meet these basic requirements will last forever and harm the service levels for your applications. It really comes down to how serious your organization is about managing change for maximum benefits. First and foremost, any change management tool must be as easy to use as possible to encourage people to use it. If that's not the case, people will circumvent the process rather than use the tool, along with the attendant risks that this brings. There's little point in issuing management edicts that threaten people who circumvent the process, as that attempts to address the symptoms and not the root cause.

It's worth considering what it means to design a usable GUI. Let's consider briefly a tool with a GUI that meets the requirements for which it was developed. One such tool is Microsoft Word, which coincidentally is the word processor I am using right now. Tools with great user interfaces have a few basic attributes: they make frequently used activities easy, make difficult ones possible, and put the user in control. To facilitate this usability, Microsoft Word displays toolbars for frequent actions, which you can even customize if your frequent activities are different from someone else's. For those less frequently used activities, you can access them via menus, and comprehensive help is available with how-to examples. When you start Microsoft Word, you are presented with a blank document. You aren't overwhelmed with a vast array of choices that are not relevant to what you are doing. In short, the user interface is not intimidating, and the user is in control, rather than the application.

You should not underestimate the many thousands of hours of user testing that Microsoft carried out in order to reach this state of graceful simplicity. Having performed a task once, a usable program will enable you to reuse the work you've done already to save time in the future. For example, in order to provide a consistent appearance for documents, Microsoft Word
provides the user with the ability to create document templates that can be reused. In the same way, you should expect your change management tool to provide features to create new changes easily by cloning an existing one to avoid duplicating previous work.

The previous discussion applies to all tools with a GUI, not just a change management tool. If users of the change management tool in your organization are frequently notified about not using the tool correctly, then the problem almost certainly lies with the user interface of the tool, and not the users. If the problem lies with the attitude of the users towards change management, then that's a management issue related to education and recruitment.

Users should never be blamed for problems with the change management process that arise because of deficiencies in the user interface design of the software used to implement the process. Statements to users like "you're not using the tool properly", should set the alarms bells ringing. Frequent users of a tool will always provide feedback on how a tool can be made more easy to use, and the ability to provide feedback should be a feature of the tool. Too often, feedback is never proactively sought, and when it is, the advice is not acted upon. However, the fact that usability problems exist actually indicates a problem with the design process for the tool itself. If you find change management in your organization a chore, you should consider how often you were requested for feedback on the usability of the tools, both at the design stage and after rollout.

Most large companies have a dedicated change management team. The role of such a team is sometimes to perform technical validation on change details. I've found that this doesn't work well in practice, because change management team members can't be expected to be professional experts in the many and various areas in which changes are required. Technical matters are best left to the technical experts, and there are minimal benefits in attempting to oversimplify changes to the point where they lose their meaning. On the other hand, change management teams can add value by chasing up change approvers to relieve the burden on the technical staff. Change management teams can also be invaluable in building up a network of application relationships to ensure that the appropriate groups are notified of changes. If the main purpose of the change management team is to report on users who aren't using the tool properly, then that's a sign that the tool is flawed and not the users' approach. Web-based tools have potential benefits in that the tool can be used from a wide range of client workstations, but they should not be used for change creation if usability is compromised. A web-based tool is definitely appropriate for reporting purposes, such as viewing forthcoming or recently implemented changes.

The fact that a ready-to-run, commercial, easy-to-use change management tool is hard to find ultimately suggests that change management does not have the emphasis that it should in most organizations.

The Change Management Process

Change management is basically a workflow application where the change being performed passes through the following states in order:

1. Creation
2. Approval
3. Implementation

4. Closure and review

In general, change management should be mandatory for changes to production systems only, in order to obtain the lowest cost and maximum benefit. At the very minimum, completing all the required information in the following section means that a process of due diligence has been followed.

Creation

During creation, the change owner creates the change using the enterprise change management tool. The change is assigned a globally unique number. The mandatory attributes of a change include the following:

- **Change owner.** The change owner should be the person who submits the change and can be contacted by any of the implementers in the case of problems. The owner is responsible for coordinating the implementation and reporting on the success or failure. That doesn't necessarily mean the owner is the person who implements the change.

- **Description.** The description should be a single-line summary of the change, suitable for presentation in a spreadsheet style list for reporting purposes. Optionally, an Extra Description should be available if the change can't be summarized meaningfully in a single line.

- **Change impact.** The change impact should identify the business and technology areas affected by the change and the scope of changes. These are useful for quantifying risk. For example, the addition of a single table to a production schema on a single database would likely impact a single application. On the other hand, the migration of an organization's Oracle Names servers onto another site represents an infrastructure change that could impact all Oracle client users.

- **Business requirement.** The requirement for the change should include a business requirement. The need for a business requirement, rather than just a simple requirement, is intended to focus on the need to avoid change for the sake of it in production systems. If a change has no business requirement, then the requirement needs to be reconsidered. More often than not, there is a business requirement underlying the need for the change. For example, a requirement to upgrade Oracle might exist because failure to upgrade will eventually result in the Oracle version falling outside the support lifecycle. In this case, the lack of support for a production problem represents a business risk. So the change has an underlying business requirement, which at first maybe isn't obvious.

- **Testing details.** The full details of testing performed related to the change should be provided. Usually, this means the full change implementation has been performed first on a development system. It's easy to make an assumption that a change, however simple, can't go wrong and therefore one fails to test it first. Such assumptions are dangerous. Consider a situation where a DBA intended to install some third-party software onto an Oracle database. The software had been installed onto many other databases already, from version 7.3.4 up to 8i, except for 8.0.4. The installation included the creation of some packages, tables, and indexes into an application schema. These operations are absolutely standard Oracle. In this case, the installation was performed on a test 8.0.4 database first, as a matter of due diligence, even though no problems were expected or foreseen. It so happened that the use of the TRUNCATE TABLE command in the software install script caused an Oracle 8.0.4-specific bug to manifest itself,
resulting in the instance crashing. The problem was not inherent in the third-party software but in the Oracle DBMS. The testing saved an outage on a business-critical production database. The lesson is that you should always test first, however straightforward the changes appear. If there is no development system to test on, then the risk of failure is higher.

**Application outage.** The need for an application outage should be flagged explicitly. Not all changes require an outage.

**Change start and end time.** The change start time and duration should be fixed at the time of change creation, because the change might contain dependencies on other groups. The duration is the time it takes to implement the change, and for the change to be operational. For example, if changes don't take effect until a database is restarted, then that would be reflected in a long duration. In this case, it would be more prudent to restart the database as part of the change implementation to reduce the duration, and avoid the risk of an error causing the database restart to fail when no immediate support is available.

**Implementation plan.** The implementation plan should contain a complete, stand-alone set of instructions that need to be followed to implement the change for all the groups involved in the implementation. For long and complex changes, the instructions might reference a separate document. For less complex changes, the instructions might include a set of SQL scripts to run. For database-related changes, full details are required regarding the database instance and account under which changes need to be applied. Assumptions should never be made.

The requirement for the instructions to be stand-alone reduces the dependency of the change on named people and forces the creation of documentation up front. In short, it enforces better planning and a higher probability of the change being complete as intended. The reason why many changes fail is incomplete planning.

**Backout plan and duration.** Full details of how to back out a change should be provided in case something goes wrong. This stage is often overlooked or includes unrealistic tasks. For example, a large data purge operation that failed might include a backout plan to restore and recover the database to the point in time before the purge. If this would mean a restore of a week-old backup and recovery taking several hours, then it might be considered impractical. The requirement for the backout plan duration is intended to flag such requirements. Long backout plan durations might require the change implementation to be modified to reduce the duration of the backout plan. In this example, a pre-purge backup might be used to reduce the backout duration at the cost of increasing the change implementation time.

**Change approvers.** All technology groups that implement a change, and all those business units whose applications are affected by a change, should be included in a list of approvers. A facility for including other groups for notification purposes is recommended.

**Priority.** At least two categories are required to include standard and emergency. Standard changes are those that require the complete list of approvers to approve the change before implementation. Emergency changes are those requiring immediate action as a result of an unscheduled business-critical outage. In this case, the change and approvals can take place retrospectively. Verbal approval at a management level of appropriate seniority should always be obtained prior to any such change being made to ensure that business managers are implicitly included in the process.

**Success/fail criteria.** Explicit success and fail criteria are required against which the change should be tested on completion.
Business impact of failure. The business impact of the change failure is required. In some cases, it may be possible to defer the change to a later time, in which case there is no business impact. In others, the change might be a one-off opportunity where failure is not an option. This is different from the requirement for a backout plan. A backout plan should always be provided and known to work. The impact of the failure of the backout plan should be considered also.

Business risk metric. Quantifying risk in a meaningful way is difficult. However, factors that influence risk can be derived from the other attributes of the change, including the complexity of the change, the requirement for an outage, the duration of the backout plan, the business impact of failure, and the contribution of the application to a company's bottom line. In general, higher risks should require a more senior level of management approval for the change.

Approval

During the approval stage, all relevant groups should signify approval of the change electronically, although approval by hard copy needs to be available as a fallback. At this stage, clarification may be sought about technical details between any of the parties involved, and this should be reflected in changes to the start time, duration, implementation, and backout plans. The change management group can add significant value at this stage by chasing up approvers on behalf of the owner and ensuring that all relevant approvers are included, possibly by incorporating extra groups affected by the change that the change owner wasn't aware of. The implementation plan should be updated to include contact details for all technical groups involved.

Once all the relevant groups have agreed to proceed, the change passes into an approved state, awaiting implementation. At this stage, no further modifications to the change are allowed. Any changes require the change to return to the created state and the requirement for approval from all parties to be sought once more. Enforcing such a policy means that more effort must be spent at the planning stage by all the groups involved, although exceptions should be allowed with appropriate management approval.

Implementation

At the implementation stage, all implementers work from the implementation details supplied in the change, and only from those details. Extra, unplanned tasks should never be included, because failure of those extra tasks isn't included in the backup plan and could take hours to unravel. The implementation contains the order and tasks for the change implementation along with the contact details of who should perform them. The change owner, as the coordinator, should be kept fully informed of progress. On completion, the owner should ensure that the success and failure criteria are checked, coordinate the backout plan if necessary, and notify the change management group of the outcome.

Closure

Once completed, the change management group moves the change to a closed state. For any cases of failure, a review should be carried out to understand the reasons for the failure in order to identify lessons learned. The review should always consider the possibility of changes to the change management process itself.

Oracle Change Management
All the attributes of the change management process in the previous section apply equally to the change management process for Oracle DBMS-based applications. This section deals specifically with the details of Oracle DBMS change management.

In general, Oracle change management means making changes to application schemas, whether installing them for the first time or carrying out upgrades or bug fixes. In order to make changes successfully, it's necessary to understand dependencies on what is being changed to assess the impact, and to ensure that the target environment being changed has the exact state expected by the change scripts. Typically, schema changes are made first in a development environment and then applied to a production schema.

Oracle and third-party vendors provide software, such as Oracle Designer and ERWin, to manage Oracle DBMS schema development. These tools can generate SQL scripts based on an application data model to install applications for the first time and generate scripts to automate the process of updating application schemas between versions.

The difficulty with this approach in the real world is that changes can be made manually to schemas outside of the data modeling tool. It’s always difficult to keep the model in step with the code. Of course, it wouldn't be if changes were only made via the modeling tool. But you can't base a production environment on such assumptions. In any case, if you're not using a fully fledged modeling tool, which requires a large investment in setup as well as ongoing maintenance and management, then you still need techniques for creating and managing schemas.

This section shows some basic techniques for the identification of differences between schemas, outlines reverse engineering object definitions, and, last but definitely not least, covers Oracle Change Manager (OCM). OCM is a tool universally admired by DBAs and developers alike for providing the capability to identify and manage Oracle schema changes for all Oracle object types with minimal effort. Perhaps unusually for an Oracle GUI-based tool, it seems to have been designed by DBAs for DBAs, and incorporates an extensive range of features with a usable graphical interface.

### Understanding Dependencies

Whenever you change an Oracle object, you potentially invalidate any object that depends on it. For example, if you decide to drop a column from a table, then any views and stored Procedural Language (PL)/SQL that references that object is invalidated. Oracle maintains dependency information between objects using information in the DBA_DEPENDENCIES view, which is defined as follows:

desc dba_dependencies

<table>
<thead>
<tr>
<th>OWNER</th>
<th>VARCHAR2(30)</th>
<th>NOT NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>VARCHAR2(30)</td>
<td>NOT NULL</td>
</tr>
<tr>
<td>TYPE</td>
<td>VARCHAR2(12)</td>
<td></td>
</tr>
<tr>
<td>REFERENCED_OWNER</td>
<td>VARCHAR2(30)</td>
<td></td>
</tr>
<tr>
<td>REFERENCED_NAME</td>
<td>VARCHAR2(64)</td>
<td></td>
</tr>
<tr>
<td>REFERENCED_TYPE</td>
<td>VARCHAR2(12)</td>
<td></td>
</tr>
<tr>
<td>REFERENCED_LINK_NAME</td>
<td>VARCHAR2(128)</td>
<td></td>
</tr>
<tr>
<td>DEPENDENCY_TYPE</td>
<td>VARCHAR2(4)</td>
<td></td>
</tr>
</tbody>
</table>
Dependency information is most naturally presented using a tree. To a limited degree, you can present the dependency information using SQL. This SQL shows the first level of dependencies on DBCOOL_SESSION_EVENT table:

```sql
REM show 1st level objects that depend on DBCOOL_SESSION_EVENT
select owner,name,type from dba_dependencies
where referenced_name='DBCOOL_SESSION_EVENT';
```

<table>
<thead>
<tr>
<th>OWNER</th>
<th>NAME</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM</td>
<td>DBCOOL_SESSION_EVENT_V</td>
<td>VIEW</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>DBCOOL_MON</td>
<td>PACKAGE BODY</td>
</tr>
</tbody>
</table>

By using the SQL CONNECT BY operation, you can display the full dependency tree using the following SQL:

```sql
select lpad(' ',l*2)||' '||object_name dep from dba_objects o,
  (
  select rownum ord,level l,d_obj#
  from sys.dependency$
  connect by prior d_obj# = p_obj#
  start with p_obj# in
  (select object_id from dba_objects
   where object_name='DBCOOL_SESSION_EVENT'
  )
  ) d
where d.d_obj#=o.object_id order by ord;
```

```
DEP
---------------------------
DBCOOL_SESSION_EVENT_V
DBCOOL_SYSTEM_EVENT_V
DBCOOL_MON
```

The SQL is somewhat unfriendly in appearance. It's not possible to query the DBA_DEPENDENCIES view directly because join conditions (as they appear in the view definition statement) can't be used in a query containing CONNECT BY. As a result, it's necessary to use the underlying SYS.DEPENDENCY$ base table instead and generate the object names for display based on the object IDs from the base table. It's also tricky to get the ordering correct to ensure that each object appears immediately below the object on which it depends. Finally, the SQL doesn't display the owner and object type in order to ensure each object fits into a 78-character line. For complex dependencies, the output can be difficult to follow.

The information is best presented graphically using a tool developed for this purpose. By right-clicking an object node in the DbCool explorer, you can present dependency information from DBA_DEPENDENCIES in a tree, showing both Depends On and Used By relationships, as shown in Figure 1.
Figure 1 Viewing dependency information using DbCool.

According to the dependency tree, the package DBCOOL_MON and the view DBCOOL_SESSION_EVENT_V use the table DBCOOL_SESSION_EVENT. The view is itself used by the view DBCOOL_SYSTEM_EVENT_V. The following code alters DBCOOL_SESSION_EVENT and, as a side effect, invalidates all objects that depend on it:

REM add column to invalidate dependents
alter table dbcool_session_event add (junk number);

REM show invalidated objects
select * from dba_objects where status <> 'VALID';

You can use DbCool to show invalid objects also by using the Object Explorer tool. Figure 2 shows the objects invalidated by adding the JUNK column including the relationships between them.
In this example, the DBCOOL_MON package body is temporarily invalid. It can be recompiled successfully following the change because the object on which it depends has been modified in a way that doesn’t actually cause a problem. If a procedure in DBCOOL_MON was executed, Oracle would transparently compile it so the invalidation could pass unnoticed. However, the two invalidated views fail to compile, because the number of columns in the table no longer matches the SELECT * in the view definitions, causing the error "ORA-01730: invalid number of column named specified." You can view the errors causing object invalidation by selecting rows for DBA_ERRORS, or alternatively you can use Code View in DbCool and double-click any object to show its code definition and any errors, as shown in Figure 3. DbCool displays a red box around invalid object names to provide a visual cue that there is a problem.

Figure 2 Viewing dependency information using DbCool.
In this case, the use of the invalid views in application code would generate an error. This example demonstrates that invalid objects may or may not present the potential for application failures: you can’t tell without checking. By being aware of the importance of dependency tracking and checking for object invalidation following changes, you can avoid production code failures due to unexpected side effects of object and code changes. To recompile invalid objects, you can use the DBMS_UTILITY.COMPILE_SCHEMA procedure. In addition, DbCool enables you to recompile invalid objects in the Object Explorer tree by right-clicking a schema name and choosing Compile Invalid from the popup menu.

It’s absolutely essential that you always check for and fix invalid objects before any change and fix any invalid objects after a change. These requirements should be included in the change success test criteria. If you don’t check for invalid objects before a change, then you can never be sure whether objects invalid after the change were caused by the change or not. This is a very dangerous situation to find yourself in, and a complete restore of the database to the pre-change state becomes a possibility.

Comparing Data Using SQL

As a DBA, there are times when you need to check the object definitions in your production database against development to see the differences between them. For example, after applying object definition changes made in development onto production as part of a change implementation, it would be useful to verify that the changes have resulted in identical schemas in both in order to verify that the change worked correctly. You can perform a fast verification using the set operations provided in Oracle’s SQL language. For example, in order to compare the table definitions in two schemas on different databases, you can use SQL to compare the metadata for the tables held in the Oracle dictionary catalog view DBA_TAB_COLUMNS. The comparison requires some basic mathematical set theory. This can be demonstrated most clearly
using a simple example based on the standard SCOTT.DEPT table as well as a copy of it, DEPT2, containing an extra column COUNTRY, and a DEPTNO column with an extra digit of precision, as shown in Table 1.

Table 1 DEPT and DEPT2 Table Descriptions

<table>
<thead>
<tr>
<th>DEPT</th>
<th>DEPT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPTNO</td>
<td>DEPTNO</td>
</tr>
<tr>
<td>NUMBER(2)</td>
<td>NUMBER(3)</td>
</tr>
<tr>
<td>DNAME</td>
<td>DNAME</td>
</tr>
<tr>
<td>VARCHAR2(14)</td>
<td>VARCHAR2(14)</td>
</tr>
<tr>
<td>LOC</td>
<td>LOC</td>
</tr>
<tr>
<td>VARCHAR2(13)</td>
<td>VARCHAR2(13)</td>
</tr>
<tr>
<td></td>
<td>COUNTRY</td>
</tr>
<tr>
<td></td>
<td>VARCHAR2(3)</td>
</tr>
</tbody>
</table>

In this example, we wish to compare the column names and data types between the two tables to identify the differences and the table where the difference is present. The SQL required to perform this operation can appear quite daunting on its own, but can be understood quite easily by building up the necessary result sets step by step and by performing set operations on them. The columns in each table can be found using the following SQL:

```sql
REM columns in DEPT
select column_name, data_type, data_length, data_precision, data_scale
from dba_tab_columns where table_name='DEPT';

REM columns in DEPT2
select column_name, data_type, data_length, data_precision, data_scale
from dba_tab_columns where table_name='DEPT2';
```

The column definitions that match between the two tables can be expressed using the set intersection operator on the column results sets for each table, using the following SQL:

```sql
select column_name, data_type, data_length, data_precision, data_scale
from dba_tab_columns where table_name='DEPT'
intersect
select column_name, data_type, data_length, data_precision, data_scale
from dba_tab_columns where table_name='DEPT2';
```

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>DATA_TYPE</th>
<th>DATA_LENGTH</th>
<th>DATA_PRECISION</th>
<th>DATA_SCALE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNAME</td>
<td>VARCHAR2</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC</td>
<td>VARCHAR2</td>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The difference between the column definitions in the two tables can be expressed using the following pseudocode that returns the columns in either table minus the columns in both tables:

```plaintext
( columns in DEPT
union
columns in DEPT2
)
minus
( columns in DEPT
union
columns in DEPT2
)
```
This can be expressed in SQL terms based on the result sets developed previously as the following:

\[
\begin{align*}
\left( 
\text{select} & \quad \text{column\_name, data\_type, data\_length, data\_precision, data\_scale} \\
& \quad \text{from dba\_tab\_columns where table\_name='DEPT'} \\
\text{union} & \\
\text{select} & \quad \text{column\_name, data\_type, data\_length, data\_precision, data\_scale} \\
& \quad \text{from dba\_tab\_columns where table\_name='DEPT2'} \\
\right) \\
\text{minus} & \\
\left( 
\text{select} & \quad \text{column\_name, data\_type, data\_length, data\_precision, data\_scale} \\
& \quad \text{from dba\_tab\_columns where table\_name='DEPT'} \\
\text{intersect} & \\
\text{select} & \quad \text{column\_name, data\_type, data\_length, data\_precision, data\_scale} \\
& \quad \text{from dba\_tab\_columns where table\_name='DEPT2'} \\
\right) \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>DATA_TYPE</th>
<th>DATA_LENGTH</th>
<th>DATA_PRECISION</th>
<th>DATA_SCALE</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNTRY</td>
<td>VARCHAR2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPTNO</td>
<td>NUMBER</td>
<td>22</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>DEPTNO</td>
<td>NUMBER</td>
<td>22</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

The result set correctly shows the list of column differences between the DEPT and DEPT2. To recap, COUNTRY is the extra column in DEPT2, and the DEPTNO column differs between the two because of the difference in number precision. The final step is to identify the source of the differences, because the output gives no indication as to which of the original tables contains the indicated columns. This information can be found for each table by the intersection of the original values in each table, with the result set of differences. In pseudocode, the columns only in DEPT can be expressed as:

\[
\text{columns in DEPT} \quad \text{intersect} \quad \text{column differences between DEPT and DEPT2}
\]

The columns only in table DEPT2 can be expressed as columns in DEPT2:

\[
\text{interact} \quad \text{column differences between DEPT and DEPT2}
\]

Finally, the complete list of differences can be expressed as the union of columns that are only in DEPT and the columns that are only in DEPT2:

\[
\left( 
\text{columns in DEPT} \\
\text{interact} \\
\right)
\]
column differences between DEPT and DEPT2
) union
(
columns in DEPT2
intersect
column differences between DEPT and DEPT2
)

The following SQL shows the first part of the result set containing columns only in DEPT:

```sql
select 1,ONLY_IN_DEPT.* from
(
select
    column_name, data_type, data_length, data_precision, data_scale
from dba_tab_columns where table_name='DEPT'
intersect
(
    
    select
        column_name, data_type, data_length, data_precision, data_scale
from dba_tab_columns where table_name='DEPT'
    union
    select
        column_name, data_type, data_length, data_precision, data_scale
from dba_tab_columns where table_name='DEPT2'
    )
  minus
  (select
      column_name, data_type, data_length, data_precision, data_scale
from dba_tab_columns where table_name='DEPT'
  intersect
      select
          column_name, data_type, data_length, data_precision, data_scale
from dba_tab_columns where table_name='DEPT2'
  )
) ONLY_IN_DEPT;
```

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>DATA_TYPE</th>
<th>DATA_LENGTH</th>
<th>DATA_PRECISION</th>
<th>DATA_SCALE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPTNO</td>
<td>NUMBER</td>
<td>22</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

The second part, showing columns only in DEPT2, is given by the following SQL:

```sql
select 2,ONLY_IN_DEPT2.* from
(
select
    column_name, data_type, data_length, data_precision, data_scale
from dba_tab_columns where table_name='DEPT2'
intersect
(
    select
        column_name, data_type, data_length, data_precision, data_scale
from dba_tab_columns where table_name='DEPT2'
    )
) ONLY_IN_DEPT2;
```
This simple example demonstrates the power of the set operators in Oracle's SQL language for identifying the differences between columns in two tables in the same database. The SQL can be modified easily to perform many other comparisons between two arbitrary data sets. For example, you could use this technique to compare the contents of two tables replicated by Oracle replication, where the contents might be out of step. The only limitations are that the column types and numbers in each set match, and the column types are restricted to those that are allowed in SQL. This excludes columns of type LONG, for example, so you couldn't use the technique to compare view definitions or trigger bodies, because those are held in LONG columns. In general, this doesn't prevent the technique from being very useful in many different situations. For example, the list of table names in two databases could be compared using the result set from the current database, with the same result set from a remote database accessed by a database link. The two result sets to compare are given by the following:

```sql
select table_name,table_owner from dba_tables

select table_name,table_owner from dba_tables@remdb.dbcool.com
```

The PL/SQL source code differences between the schema PROD in the current database and the schema DEV in another could be compared using the following result sets:

```sql
select name,type,line,text from dba_source where owner='PROD'

select name,type,line,text from dba_source@remdb where owner='DEV'
```
The init.ora parameters between two related sites, such as sites in a RAC cluster or sites in a multimaster replication configuration, could be compared using the following SQL result sets:

```
select name, value from v$parameter@site1.dbcool.com
```

```
select name, value from v$parameter@site2.dbcool.com
```

In this last example, you might expect to find differences related only to the `db_name`, `global_name`, and `service_names` parameters. Any others would probably indicate accidental differences between the sites that need to be fixed. Based on the `DEPT` and `DEPT2` example, the SQL to perform the comparison in all the previous examples is straightforward if somewhat tedious to produce. DbCool provides a built-in feature to perform a comparison of two arbitrary data sets. To use the feature, you first need to add the object to compare into the Object Explorer tree using the Object Finder tool. Next, right-click the object node in the tree, and choose Compare Data With, which displays the form shown in Figure 4.

**Figure 4 Comparing table definitions using DbCool.**

The result sets to compare can be different tables on the same database, on different databases, or even different result sets in the same table. When the Run Compare button is pressed, the result set showing the differences is merged for easy assimilation and displayed in the grid. Figure 5 shows some output based on the `DEPT` and `DEPT2` example presented earlier.

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>DATA_TYPE</th>
<th>DATA_LENGTH</th>
<th>DATA_PPCISION</th>
<th>DATA_SCALE</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNTRY</td>
<td>VARCHAR2</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>compare</td>
</tr>
<tr>
<td>DEPTNO</td>
<td>NUMBER</td>
<td>22</td>
<td>3</td>
<td>0</td>
<td>source</td>
</tr>
</tbody>
</table>

**Figure 5 Table differences report using DbCool.**

**Visual Presentation of Differences**

Most Oracle applications of any significance contain thousands of lines of PL/SQL packaged code. The SQL-based approach presented previously can be used for identification of differences between object definitions across different systems, including PL/SQL. Due to the row-based nature of the output, it's probably more useful for checking result sets where no differences, or just a few, are expected. It's a useful technique for confirming that result sets are the same. A different technique is required to present differences in a more meaningful way.

If you want to compare PL/SQL code differences between versions of a package or procedure, or between different sites, then DbCool has a feature to compare them. In order to show an example of the feature, it's necessary to have Microsoft Word installed, because DbCool takes advantage of Word's change-tracking features to perform the comparison in a highly visual way that's appropriate to the task. As a simple example, consider a package on a production system that presents a PL/SQL interface to the EMP table as follows:

```
create or replace package plsql_emp as
    function sal_by_dept(p_dname varchar2) return number;
end plsql_emp;
/```
create or replace package body plsql_emp as
    function sal_by_dept(p_dname varchar2) return number is
        l_sum number;
        begin
            select sum(sal) into l_sum from emp where deptno=
                 (select deptno from dept where dname=p_dname);
            return l_sum;
        end;
end plsql_emp;
/

Consider a situation where, at a later stage, a new development manager takes over and
decides that the code requires some comments and an exception handler. The following changes
are made on the development server:

create or replace package plsql_emp as
    ------ return sum of salary in named dept
    function sal_by_dept(p_dname varchar2) return number;
end plsql_emp;
/

create or replace package body plsql_emp as
    function sal_by_dept(p_dname varchar2) return number is
        l_sum number;
        begin
            select sum(sal) into l_sum from emp where deptno=
                 (select deptno from dept where dname=p_dname);
            return l_sum;
            exception
                when others then
                    raise_application_error(-20001,
                        'DEPT='||p_dname||sqlerrm);
            end;
        end;
end plsql_emp;
/

You can compare the two versions with Microsoft Word using DbCool. First, you need
to add the original package to the DbCool Object Explorer. Next, you right-click the package
body node, choose Compare Definition With, and enter the database, schema, and name of the
object that you want to compare. The two versions are passed to Microsoft Word, which tracks
changes between the two versions in such a way that differences stand out clearly, as shown in
Figure 6.
In general, the comparison of differences between any Oracle object definitions benefits from this technique. The presentation of the differences adds meaning. For the case of table definitions, it would be useful to perform the same visual comparison and include a feature to ignore differences between the storage clauses and other physical attributes that aren't of interest. These are the kinds of features you would expect from a commercial change management tool, along with the capability to create Data Definition Language (DDL) scripts both to create new application schemas from scratch and to update the current production schema with changes made in development. Manual processes to perform these tasks are prone to error and likely to result in application outages.

Reverse Engineering Object Definitions

In order to deliver an application schema from a development database to a production environment, a script is required to install the application objects. In theory, if you use a data modelling tool, you should be able to generate the object creation script using the tool. The concern when generating scripts using a modelling tool is that changes made manually outside the tool might be missed, resulting in a model that's out of date. Perhaps you don't use a modelling tool in any case. At the same time, it's usually the case that whatever definitions exist in the development database schema are exactly correct, because that schema was used for all the user acceptance testing. In this case, one approach that can be used to generate the schema is to reverse engineer the DDL creation statements of all the objects from the development database data dictionary.

Oracle9i provides a package, DBMS_METADATA, which can be used to reverse engineer object definitions. The simplest interface for returning object definitions is the GET_DDL procedure, which returns a CLOB datatype. The following SQL returns the object definition for SCOTT's EMP table:

```
select dbms_metadata.get_ddl('TABLE', table_name, owner) 
from all_tables where owner='SCOTT' and table_name='EMP';
```

```sql
CREATE TABLE "SCOTT"."EMP"  
( "EMPNO" NUMBER(4,0),  
  "ENAME" VARCHAR2(10),  
  "JOB" VARCHAR2(9),  
  "MGR" NUMBER(4,0),  
  "HIREDATE" DATE,  
  "SAL" NUMBER(7,2),  
  "COMM" NUMBER(7,2),  
  "DEPTNO" NUMBER(2,0),  
  CONSTRAINT "PK_EMP" PRIMARY KEY ("EMPNO")  
USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255  
STORAGE(INITIAL 16384 NEXT 16384 MINEXTENTS 1 MAXEXTENTS 5  
PCTINCREASE 0  
FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT) TABLESPACE "TOOLS" ENABLE,  
  CONSTRAINT "FK_DEPTNO" FOREIGN KEY ("DEPTNO") REFERENCES "SCOTT"."DEPT" ("DEPTNO") ENABLE NOVALIDATE  
) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING
```
The full SQL of the table definition is returned by default, including primary and unique key constraints, and foreign key references inline. As another example, the following SQL can be used to reverse engineer all index definitions in SCOTT's schema:

```sql
select dbms_metadata.get_ddl('INDEX', index_name, owner) from all_indexes where owner='SCOTT';
```

```
CREATE UNIQUE INDEX "SCOTT"."PK_DEPT" ON "SCOTT"."DEPT" ("DEPTNO") PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE(INITIAL 16384 NEXT 16384 MINEXTENTS 1 MAXEXTENTS 5 PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT) TABLESPACE "TOOLS"
```

```
CREATE UNIQUE INDEX "SCOTT"."PK_EMP" ON "SCOTT"."EMP" ("EMPNO") PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE(INITIAL 16384 NEXT 16384 MINEXTENTS 1 MAXEXTENTS 5 PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT) TABLESPACE "TOOLS"
```

These simple examples don't do justice to the full capabilities of DBMS_META, although most Oracle DBAs would probably gasp in surprise at seeing object definitions created from SQL queries for the first time. For example, the original EMP creation statement isn't in the most suitable format for incorporating into an installation script. Rather than including all index and constraint creation inline with CREATE TABLE, it's much more flexible for a schema creation script to create tables first without any constraints, then to add primary keys in a separate pass, and finally to add foreign keys.

During bulk object creation scripts, constraint and foreign key creation order is important in order to avoid object creation failures caused by references to dependent objects that don't exist yet. In the EMP example, the inline definition of the foreign key that references SCOTT.DEPT would cause the EMP creation to fail, unless DEPT and its primary key existed already. Using such inline definitions for primary key and foreign constraints is impractical as it places too much emphasis on the DBA to get the order correct. Also, the EMP definition includes storage details such as the reference to the tablespace TOOLS. For the most flexible installations, the choice of tablespace should be an installation parameter and not coded inline. Storage parameters have limited use in any case when local uniform space allocation and automatic segment management are in use. By using the full capabilities of DBMS_METADATA, both of these problems can be solved: tables can be created without the inline constraints and without storage and tablespace clauses. The suppression of storage and tablespace clauses can be suppressed individually or together for the current session through the following package call:

```
REM suppress generation of segment storage
begin
```
After disabling storage attributes for the session, SQL used to generate SCOTT’s tables produces much simpler output:

```
CREATE TABLE "SCOTT"."DEPT"
  ( "DEPTNO" NUMBER(2,0),
    "DNAME" VARCHAR2(14),
    "LOC" VARCHAR2(13),
    CONSTRAINT "PK_DEPT" PRIMARY KEY ("DEPTNO") ENABLE
  )

CREATE TABLE "SCOTT"."EMP"
  ( "EMPNO" NUMBER(4,0),
    "ENAME" VARCHAR2(10),
    "JOB" VARCHAR2(9),
    "MGR" NUMBER(4,0),
    "HIREDATE" DATE,
    "SAL" NUMBER(7,2),
    "COMM" NUMBER(7,2),
    "DEPTNO" NUMBER(2,0),
    CONSTRAINT "PK_EMP" PRIMARY KEY ("EMPNO") ENABLE,
    CONSTRAINT "FK_DEPTNO" FOREIGN KEY ("DEPTNO")
    REFERENCES "SCOTT"."DEPT" ("DEPTNO") ENABLE NOVALIDATE
  )
```

However, constraint definitions still appear inline with the table definition, something we want to avoid. An additional modification can be made to reverse engineer the constraints as separate ALTER TABLE statements, rather than inline with the table creation, by using the CONSTRAINTS_AS_ALTER transform. In addition, the SQLTERMINATOR transform can be used to terminate each statement with either ‘;’ or ‘/’ to make the generated output suitable for executing in SQL*Plus. These transforms are activated as follows to produce the output shown:

```
REM generate constraints out of line, as ALTER TABLE statements
begin
  dbms_metadata.set_transform_param(
    transform_handle=>dbms_metadata.session_transform,
    name=>'CONSTRAINTS_AS_ALTER',
    value=>true);
end;
/
begin
  dbms_metadata.set_transform_param(
    transform_handle=>dbms_metadata.session_transform,
    name=>'SQLTERMINATOR',
    value=>true);
end;
```
REM REPLACE used to split lines makes output more readable...
select replace(dbms_metadata.get_ddl('TABLE', table_name, owner),
               'ADD CONSTRAINT','ADD CONSTRAINT'||chr(10)||' ')
from all_tables where owner='SCOTT' and table_name='EMP';

CREATE TABLE "SCOTT"."EMP"
  ( "EMPNO" NUMBER(4,0),
    "ENAME" VARCHAR2(10),
    "JOB" VARCHAR2(9),
    "MGR" NUMBER(4,0),
    "HIREDATE" DATE,
    "SAL" NUMBER(7,2),
    "COMM" NUMBER(7,2),
    "DEPTNO" NUMBER(2,0)
  );
CREATE UNIQUE INDEX "SCOTT"."PK_EMP" ON "SCOTT"."EMP" ("EMPNO");
ALTER TABLE "SCOTT"."EMP" ADD CONSTRAINT
  "PK_EMP" PRIMARY KEY ("EMPNO") ENABLE;
ALTER TABLE "SCOTT"."EMP" ADD CONSTRAINT
  "FK_DEPTNO" FOREIGN KEY ("DEPTNO")
  REFERENCES "SCOTT"."DEPT" ("DEPTNO") ENABLE NOVALIDATE;

Although the constraints are now created using separate statements, the previous output is actually returned in a single row for each table. Newline characters are embedded to make the output readable. That's fine for casual inspection, but not especially useful if the requirement is to split the output into separate scripts for table, index, constraint, and foreign key creation.

In order to meet this requirement, it's necessary to stop using SQL queries to generate the DLL and call the package procedures in DBMS_METADATA directly from within PL/SQL instead. The package contains additional procedures to parse the output row for each table and pull out the individual DDL statements in each row. The UTL_FILE package is available to write the output to separate files. The following PL/SQL code shown in Code Figure 1 contains a complete and self-contained example, based on the EMP and DEPT table, that demonstrates many of the capabilities of DBMS_METADATA for generating SQL DLL scripts in the right order.

NOTE
The utl_file_dir database initialization parameter must include /tmp for the following example to work correctly.
table_ddls sys.ku$_ddls; -- DDL lists for each table
table_ddl sys.ku$_ddl; -- single DDL statement

parsed_items sys.ku$_parsed_items;

---- output files follow...
f_tab utl_file.file_type; f_ind utl_file.file_type; 
f_con utl_file.file_type; f_fk utl_file.file_type;
f utl_file.file_type;
d number;
begin

-- !!! Note: /tmp must be in utl_file_dir param in init.ora

f := utl_file.fopen('/tmp','tab.sql','w',32760); f_tab := f;
f := utl_file.fopen('/tmp','ind.sql','w',32760); f_ind := f;
f := utl_file.fopen('/tmp','con.sql','w',32760); f_con := f;
f := utl_file.fopen('/tmp','fk.sql','w',32760); f_fk := f;

h_table := dbms_metadata.open('TABLE');

-- set items we want to identify in each DDL
dbms_metadata.set_parse_item(h_table,'SCHEMA');
dbms_metadata.set_parse_item(h_table,'NAME');
dbms_metadata.set_parse_item(h_table,'VERB');
dbms_metadata.set_parse_item(h_table,'OBJECT_TYPE');

-- request information to be returned as SQL DDL
hx_table := dbms_metadata.add_transform(h_table,'DDL');

-- 1) suppress tablespace and storage attributes
-- 2) add SQL*Plus terminator to SQL
-- 3) return constraints as separate ALTER TABLE statements

dbms_metadata.set_transform_param(hx_table,
   'SEGMENT_ATTRIBUTES',FALSE);

dbms_metadata.set_transform_param(hx_table,
   'SQLTERMINATOR',TRUE);

dbms_metadata.set_transform_param(hx_table,
   'CONSTRAINTS_AS_ALTER',TRUE);

loop
' get list of DDLs for each table...
table_ddls := dbms_metadata.fetch_ddl(h_table);
exit when table_ddls is null;

' for each DDL in a single table...
for d in 1..table_ddls.count loop
table_ddl := table_ddls(d);

parsed_items := table_ddl.parseditems;

for i in 1..4 loop -- get parsed items out of DDL
  if parsed_items(i).item = 'SCHEMA' then
    schema := parsed_items(i).value;
  elsif parsed_items(i).item = 'NAME' then
    object_name := parsed_items(i).value;
  elsif parsed_items(i).item = 'VERB' then
    verb := parsed_items(i).value;
  elsif parsed_items(i).item = 'OBJECT_TYPE' then
    object_type := parsed_items(i).value;
  end if;
end loop;

-- table_ddl.ddltext contains DDL.
-- now work out the DDL type, and put in the correct file

if verb='CREATE' and object_type = 'TABLE' then
  utl_file.put_line(f_tab,table_ddl.ddltext);
  utl_file.fflush(f_tab);
elsif verb='CREATE' and object_type='INDEX' then
  utl_file.put_line(f_ind,table_ddl.ddltext);
  utl_file.fflush(f_ind);
elsif verb='ALTER' and instr(table_ddl.ddltext, ' FOREIGN KEY ') > 0 then
  utl_file.put_line(f_fk,table_ddl.ddltext);
  utl_file.fflush(f_fk);
elsif verb='ALTER' and
  instr(table_ddl.ddltext,' ADD CONSTRAINT ') > 0 then
  utl_file.put_line(f_con,table_ddl.ddltext);
  utl_file.fflush(f_con);
else
  dbms_output.put_line('UNKNOWN SQL:'||table_ddl.ddltext);
end if;
end loop;
end loop;

-- close files
utl_file.fclose(f_tab);utl_file.fclose(f_ind);
utl_file.fclose(f_fk);utl_file.fclose(f_con);

end;
/

/*
The script produces four output files on the database server in the /tmp directory: tab.sql, ind.sql, cons.sql, and fk.sql. The contents of the files are as follows:
*/

REM ***** tab.sql *****
CREATE TABLE "SCOTT"."DEPT"
  (   "DEPTNO" NUMBER(2,0),
"DNAME" VARCHAR2(14),
"LOC" VARCHAR2(13)
); 

CREATE TABLE "SCOTT"."EMP"
(
  "EMPNO" NUMBER(4,0),
  "ENAME" VARCHAR2(10),
  "JOB" VARCHAR2(9),
  "MGR" NUMBER(4,0),
  "HIREDATE" DATE,
  "SAL" NUMBER(7,2),
  "COMM" NUMBER(7,2),
  "DEPTNO" NUMBER(2,0)
); 

REM ***** ind.sql *****
CREATE UNIQUE INDEX "SCOTT"."PK_DEPT" ON "SCOTT"."DEPT"
("DEPTNO")
;
CREATE UNIQUE INDEX "SCOTT"."PK_EMP" ON "SCOTT"."EMP" ("EMPNO")
;

REM ***** cons.sql *****
ALTER TABLE "SCOTT"."DEPT" ADD CONSTRAINT "PK_DEPT" PRIMARY KEY
("DEPTNO") ENABLE;

ALTER TABLE "SCOTT"."EMP" ADD CONSTRAINT "PK_EMP" PRIMARY KEY
("EMPNO") ENABLE;

REM ***** fk.sql *****
ALTER TABLE "SCOTT"."EMP" ADD CONSTRAINT "FK_DEPTNO" FOREIGN
KEY ("DEPTNO") REFERENCES "SCOTT"."DEPT" ("DEPTNO") ENABLE NOVALIDATE;

---

**Code Figure 1 Reverse engineering with DBMS_METADATA.**

These scripts can be run in the order shown, in order to create a simple application schema that will install first time:

- tab.sql
- ind.sql
- cons.sql
- fk.sql

The order of creation is the same one that Oracle's Import utility uses to ensure that dependencies between the objects don't cause problems. This is exactly what is required.
It's useful to compare the features of DBMS_META with third-party tools that provide similar features. Third-party tools to reverse engineer schema definitions are typically expensive and prone to error. For example, the Oracle CREATE TABLE statement now runs to several pages in the manual, and it's very difficult for third-party software developers to get the DDL syntax exactly right for all situations. I speak from first-hand experience on this one, as DbCool itself contains many object reverse-engineering features performed by manually coding the results that DBMS_METADATA generates in a few package calls.

If you want to consider a different approach, Oracle Export dumps can be used as a basis for generating the DLL, because they contain DLL statements for all the objects contained in the dump. You might want to take a look at the extract_dll.pl Perl utility available on Oracle Metalink [http://metalink.oracle.com](http://metalink.oracle.com) that can parse a binary dump file and extract many different types of statements; it is not just restricted to tables. Hopefully over time, DBMS_METADATA will evolve to support reverse engineering of all Oracle objects, such as tablespaces and redo logs.

Today DBMS_METADATA can solve most of your SQL DDL code generation problems. The previous single-page PL/SQL block example forms a basis for a complete reverse-engineering tool. It's not complete, as it doesn't cover all the required types, such as packages. Also, the code assumes the DDL returned is fairly small. In an industrial-strength implementation, the code would need to handle DDL returned a CLOB values - potentially many thousands of bytes long - such as those resulting from tables with hundreds of partitions. Error handling would also need to be tightened up. On a more positive note, DBMS_METADATA is huge step forward in Oracle's reverse-engineering capabilities, as it puts reliable and robust DDL generation facilities directly into the database itself. It also provides third-party software product developers with the ability to generate 100-percent-correct installation scripts for their applications, which benefits the reliability of business applications based on their products.

**Oracle Change Manager (OCM)**

OCM is a GUI tool that provides the ability to automate the management of schema changes. You begin by creating a named baseline snapshot of an application schema containing all objects or a subset that you choose manually. At a later time, you can save another snapshot, view changes between the snapshots, create a SQL DDL script of differences to perform schema upgrades and propagate the changes. In short, OCM contains all the functionality you need to manage Oracle schema changes within a single tool through a very well-designed GUI. I sometimes find Oracle's GUI tools frustrating to use, based on usability expectations set by using Microsoft's tools. However, OCM is a pleasure to use, because it presents the features in the way you would expect. On UNIX, OCM is started using the following command line and requires an X Windows display:

```
$ oemapp ocm
```

Like most other OEM tools, OCM works best when used with a Management Server and OEM Repository for capturing the schema information. Figure 7 shows the OCM main screen containing the icons that represent the main components.
As usual, the functionality of OCM is best understood using an example. In this example, the goal is to manage changes for SCOTT's schema between version one and version two of the schema. Version one contains the following objects:

```sql
create table dept
    (deptno number(2) constraint pk_dept primary key,
     dname varchar2(14),
     loc varchar2(13) );

create table emp
    (empno number(4) constraint pk_emp primary key,
     ename varchar2(10),
     job varchar2(9),
     mgr number(4),
     hiredate date,
     sal number(7,2),
     comm number(7,2),
     deptno number(2) constraint fk_deptno references dept);
```

Change Manager and its integrated tools allow you to:
- Capture DB object definitions in a "baseline".
- Compare and synchronize DB object definitions.
- Search for objects by name.
- Change DB object definitions at one or more databases.

Click the icons on the right for a brief description of each tool, or see Choose a Change Tool.

Figure 7 OCM main screen.

For more information, see the Quick Tour.
create table salgrade
(
grade number,
losal number,
hisal number
);

create or replace package plsql_emp as
    function sal_by_dept(p_dname varchar2) return number;
end plsql_emp;
/

create or replace package body plsql_emp as
    function sal_by_dept(p_dname varchar2) return number is
        l_sum number;
    begin
        select sum(sal) into l_sum from emp where deptno=
            (select deptno from dept where dname=p_dname);
    end;
    return l_sum;
end plsql_emp;
/

In order to store object details at the point in time of version one, it's necessary to create a baseline from the main menu of OCM. The baseline stores a persistent representation of the schema in the OEM repository database and associates it with a user-chosen name. Figure 8 shows the options available when creating a baseline.
Figure 8 OCM baseline creation.

It's possible to select all schemas in the database, along with database objects as well as user objects, and to include grants and database parameters settings. In the example, only SCOTT's objects are required. The baseline schema information is saved into the OEM Repository with a name you choose, such as SCOTT Version 1. Figure 28-9 shows the information captured in the baseline.
In version two of the schema, a trigger and sequence are added to automatically generate the primary key column for EMP. Additionally, an exception handler and comments are added to the package, and the size of columns in the DEPT and EMP table is increased in width as follows:

```sql
alter table scott.dept modify(deptno number(3));

alter table scott.emp modify (empno number(5));
```

After changes have been made to the database, they can be saved as a new baseline, such as SCOTT Version 2. Once saved, changes can be compared between base lines, side by side onscreen using the Compare Database Objects Wizard in the main Objects menu. OCM also supports comparisons between two databases as well as baseline/schema combinations.

OCM has a very extensive range of comparison options. For example, you can ignore the physical storage attributes of segments in the comparison and ignore tablespaces. For constraint definitions, you can choose to match them by definition or by name. By definition is useful if the schema uses Oracle-generated names for constraints (named SYS) rather than user-chosen names. Allowing Oracle to choose the names represents questionable design, because it means the EMP primary key constraint will probably have a different name in different databases. Oracle works fine in this situation, but it makes the identification of differences more difficult. In this example, the EMP primary key has the name PK_EMP, as chosen by the user, so it's not a problem. The good news is that OCM can handle both cases. Figure 28-10 shows the Comparison Options screen.
Figure 10 OCM Comparison Options.

Changes can be viewed after the comparison is complete through a high-level list and at the object level for each object that has changed. The high-level list shows objects that exist in both versions with and without changes, and those that exist in one version or the other as shown in Figure 11.

<table>
<thead>
<tr>
<th>State</th>
<th>Type</th>
<th>Schema</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑ ☑</td>
<td>PACKAGE</td>
<td>SCOTT</td>
<td>PLSQL_EMP</td>
</tr>
<tr>
<td>☑  ☑</td>
<td>PACKAGE BODY</td>
<td>SCOTT</td>
<td>PLSQL_EMP</td>
</tr>
<tr>
<td>☐</td>
<td>SEQUENCE</td>
<td>SCOTT</td>
<td>EMP_S</td>
</tr>
<tr>
<td>☑ ☑</td>
<td>TABLE</td>
<td>SCOTT</td>
<td>DEPT</td>
</tr>
<tr>
<td>☑ ☑</td>
<td>TABLE</td>
<td>SCOTT</td>
<td>EMP</td>
</tr>
<tr>
<td>☑ ☑</td>
<td>TABLE</td>
<td>SCOTT</td>
<td>SALGRADE</td>
</tr>
<tr>
<td>☐</td>
<td>TRIGGER</td>
<td>SCOTT</td>
<td>EMP_I</td>
</tr>
</tbody>
</table>

Figure 11 OCM differences list.

Figure 12 shows the information presented after drilling down to the differences in the PLSQL_EMP package body, and it shows lines that have been added side by side with the original code. The figure really needs a color display, because the deleted lines are shown in blue, the changed lines in red, and the inserted lines in green. The added lines are visible in the right-hand pane in green. You'll have to take my word for that.
The Synchronization Wizard from the Tools menu can be used to generate a SQL script to synchronize a schema with a baseline. However, although it's not possible to use the Synchronization Wizard to generate SQL DDL differences between two baselines, in reality this isn't an issue.

For the sake of example, assume that baseline SCOTT Version 1 refers to a schema definition that exists in the production database, instance ORAP1, and that baseline SCOTT Version 2 is generated from the development database ORAD1. The SCOTT Version 2 schema includes the changes that have been made during the development lifecycle to enhance the application.

In this case, you can use Compare Database Objects from the Objects menu to compare the SCOTT schema in ORAP1 with the new development schema represented by the baseline SCOTT Version 2. The Synchronization Wizard generates a change plan that you need to provide a name for. A good choice of name would be the internal change ID generated by your change management system. The change plan can contain all or a subset of the objects that have changed, according to your requirements. In this case, all the changed objects are chosen. The change plan can then be stored persistently, and executed immediately or deferred until later. Manual changes to the generated plan are supported. As part of the generation of the change plan, an invaluable impact analysis report is generated to warn of possible problems that might arise as a result of the changes, including changes that won't be undone automatically if things go wrong. Figure 13 shows part of the generated script for moving the changes across to production, including the column modifications referred to earlier.
Figure 13 OCM Synchronization SQL.

Provided that the script has generated successfully, it can be saved and then optionally executed immediately, or submitted as a database job. You should revisit the impact analysis report before you go ahead and execute. Should execution fail, an undo option is available, but be warned that some changes can't be undone, and these are included in the impact analysis report. The execution log contains the result of execution, and part of the log for this example is shown in Figure 14.
Summary

Change management requires software with an excellent user interface. If the change management process in your organization is viewed as obstructing real work, then the process and tools used should be carefully reappraised. Every change process must include a complete implementation plan, a realistic backout plan, a thorough impact analysis, and clear change success criteria.

Too often, the change management process is flawed at the outset due to the non-availability of suitable tools to implement it. In this case, the process is at fault and not the people that use it. A well-designed change management tool acts as a centralized inventory for all changes, allowing trends in change volumes and change failures to be identified to ensure that things like QA and testing procedures are being followed.

The impact analysis for an Oracle change requires an understanding of Oracle dependencies and tools for identifying differences between schemas. The delivery of code from development to production requires the generation of DDL. Identifying differences can be performed using SQL, and the Oracle9i DBMS_METADATA package makes DDL generation easier than ever before. For industrial-strength management of Oracle changes, you need look no further than the OCM tool, which provides most of the features that DBAs need to manage change within an easy-to-use GUI. As Oracle object definitions become increasingly complex over time, the process of managing change is best left to Oracle rather than third-party tools to have the highest chance of success.