CHAPTER 13

Tuning with Optimizer
Plan Stability
This chapter discusses the use of optimizer plan stability in Oracle8i and shows how you can improve the run-time performance of SQL statements and also provide an easy method to permanently change the execution plans for SQL statements. This chapter will cover the following topics:

- Introduction to stored outlines
- Preparing Oracle for stored outlines
- How to create and modify a stored outline
- Managing a stored outline

**Introduction to Stored Outlines**

The optimizer plan stability feature of Oracle8i has been a long time coming. The earlier databases such as DB2 and IDMS have had the ability to store execution plans since the 1980s, and the concept of stored SQL outlines has had widespread acceptance in the non-Oracle world for decades.

The argument behind optimizer plan stability is that there exists only one optimal execution plan for any SQL statement, and once located, the execution plan should never change, even when the CBO statistics or initialization parameters are changed. Of course, this philosophy is contrary to the basic tenet of Oracle cost-based optimization, which expects SQL statements to change execution plans when the characteristics of the table and index statistics change, or when a change is made to an important initialization parameter such as `sort_area_size` or `db_file_multiblock_read_count`.

Regardless of philosophy, creating a stable execution plan for Oracle SQL has two major benefits:

- **Change execution plan without touching SQL source code**  Many databases have SQL that is dynamically generated (e.g., SAP) or SQL that resides in unreachable PC libraries. For these types of applications, stored outlines allow you to change the execution plan for the SQL without the need to change the SQL source code.

- **Permanent SQL tuning changes**  Once tuned, optimizer plan stability allows for SQL statements to always have the same execution plan. There will be no surprises when a change is made to an important initialization parameter such as `sort_area_size` or when the CBO statistics change.
Stored outlines are great for tuning SQL in database application suites where the SQL source is not available or cannot be changed. For example, SAP and PeopleSoft applications products can now have SQL tuning without touching the source code.

Now that we see the benefits of using stored outlines, let’s take a look at how optimizer plan stability works. When a SQL statement enters Oracle8i, the database will perform the following actions (Figure 13-1).

1. **Check shared pool** Hash the SQL statement and see if an identical statement is ready to go in the shared pool. If it is found, reexecute the SQL statement from the shared pool.

2. **Check stored outlines** If the SQL is not found in the shared pool, check for a stored outline in DBA_OUTLINES view in the OUTLINE tablespace. If a stored outline is found, load it into the shared pool and begin execution.

3. **Start from scratch** If nothing for the SQL statement is found in the shared pool or stored outlines, parse the SQL, develop an execution plan, and begin execution.

**FIGURE 13-1.** Executing a new SQL statement
Optimizer plan stability can also aid us in our SQL tuning effort. Oracle provides a package called `outln_pkg` that allows you to manage your stored outlines and Oracle also provides an option for the `ALTER SYSTEM` and `ALTER SESSION` commands for the automatic collections of stored outlines for all of your SQL statements. However, it is not a good idea to turn-on the option to automatically store outlines unless you are prepared to examine each stored outline and ensure that the best execution plan is being used.

If you have already tuned your SQL, you can `alter system set_stored_outlines=true` and Oracle will define plans for all issued SQL statements in all sessions at the time they are executed; these stored plans will be reused until altered or dropped. If you have not tuned your SQL, you can create individual stored outlines for your session by entering the `alter session set_stored_outlines=true` command.

Just as the shared pool will not reuse a SQL statement unless it is identical to a previous statement, the effective use of stored outlines depends on the statement being reissued in an identical fashion each time it is used. If even one space is out of place, the stored outline is not reused. Whenever possible, all SQL should be stored in PL/SQL procedures, functions, or packages, and bind variables should always be used to ensure that the SQL is reusable. This allows reuse of the stored image of the SQL as well as reuse of your stored outlines.

**The Philosophy Behind Optimizer Plan Stability**

The use of stored outlines depends heavily upon the nature of the SQL that you have in your shop. If your system is characterized by dynamic tables where the number of rows may change daily from 1,000,000 to 100, then you would probably not want to use optimizer plan stability, because you would want the CBO to reevaluate the execution plan for each SQL statement based on the current CBO statistics.

On the other hand, if your tables are relatively stable, using stored outlines will greatly improve the performance of your SQL because the SQL parse phase is bypassed. You also have the peace of mind of knowing that a SQL statement will always use the same execution plan regardless of changes to Oracle initialization parameters and CBO statistics.

Remember, there are many DBAs who feel that for any SQL statement there exists only one optimal execution plan, and once located, it should never change. If you are one of these DBAs, then stored outlines can greatly aid your SQL tuning effort.

If you choose to implement stored outlines for all of your SQL, you must also remember that CBO statistics will only be used by new queries. Hence, you may want to reduce the frequency of your reanalysis of tables and indexes.

The use of stored outlines also aids you when you are migrating from rule-based to cost-based optimization. While you begin to tune SQL statements to use the CBO, you can use optimizer plan stability to ensure that your other SQL statements continue to use their rule-based execution plans. Let’s take a look at how this works.
Using Optimizer Plan Stability When Migrating to the CBO

Because Oracle8i is the first release of Oracle where the CBO has comparable execution speeds to the RBO, many DBAs are looking at methods for safely migrating their SQL into a database with a cost-based optimizer default. If your application was developed using the rule-based optimizer, then you may have already invested a considerable amount of effort into manually tuning the SQL statements to optimize performance.

Most RBO applications use a rule-based default optimizer mode (optimizer_mode=rule) and then tune specific SQL queries with cost-based hints. While the tuned SQL has hints to ensure fast execution speeds, the majority of the SQL using the rule-based default may change execution plans when migrated into a database with a cost-based optimizer default (optimizer_mode=first_rows). You can use optimizer plan stability to preserve your execution plans when upgrading from rule-based to cost-based optimization.

By creating stored outlines for every SQL statement before switching to cost-based optimization, you can use the plans generated by the rule-based optimizer, while statements generated by newly written applications developed after the switch use cost-based plans. To create and store outlines for your system before migrating to the CBO, use the alter system set use_stored_outlines=true command. This will create an execution plan for every SQL statement.

Later, when you are ready to see if your nonhinted rule-based SQL statements will run faster with the Oracle8i CBO, you can run a query against the DBA_OUTLINES to identify those SQL statements that do not contain hints and tune them individually using the procedures described later in this chapter.

Next, let’s get started and see how to prepare your database to allow optimizer plan stability.

Preparing Oracle for Stored Outlines

Several steps need to be completed before Oracle will use stored outlines. These include:

- Setting the use_stored_outlines command
- Verifying important initialization parameters
- Creating the outline package
- Defining the tablespace for the stored outlines
Setting the use_stored_outlines Command

Oracle provides a parameter called use_stored_outlines to enable optimizer plan stability. The use_stored_outlines parameter is not an initialization parameter, although it probably will become an initialization parameter in Oracle 9i. To enable the use of stored outlines for the whole database, you can issue the following command:

```
alter system set use_stored_outlines=true;
```

However, a new parameter called USE_PRIVATE_OUTLINES enables the use of a category of outlines strictly for the current session while other sessions may use a different set. In addition 9i OEM has a outline editor to facilitate changing the outlines.

Of course, this command must be reissued each time the database is restarted. Most Oracle DBAs add this command to the post-startup procedure that executes dbms_shared_pool.keep to pin Oracle packages into the shared pool.

You can also enable optimizer plan stability at the session level by issuing the alter session set use_stored_outlines=true command, but it is much more effective to enable stored outlines at the database level, provided that your system does not generate SQL with embedded literal values.

CAUTION

The outline users default system tablespace can become exhausted if the alter system set create_stored_outlines=true and your application issues lots of nonreusable SQL with literal values. If this happens, set cursor_sharing=force in your initialization file and use the drop_unused function of the outline package to remove the unused outlines.

Verifying Important Initialization Parameters

Oracle recommends several initialization parameters that must be set in order to use stored outlines. These parameters include:

- **cursor_sharing=force**    Set this parameter if your system uses SQL with embedded literal values. With cursor_sharing set, the SQL will be rewritten to replace the literals with host variables, making the stored outlines reusable.

- **query_rewrite_enabled=true**    This enables materialized views.
Chapter 13: Tuning with Optimizer Plan Stability

- **star_transformation_enabled=true** This enables the star join.
- **optimizer_features_enable=true** This will enable the initialization parameters `b_tree_bitmap_plans`, `complex_view_merging`, `fast_full_scan_enabled`, and `push_join_predicate`.

Create the Outline Package

You will need to install the `outln_pkg` package to get the procedures available to maintain your stored outlines. Upon installation of the Oracle8i software, a user named `outln` is created with a password of `outln`. The script to create the `outln_pkg` packages is called `dbmsol.sql` in the `$ORACLE_HOME/rdbms/admin` directory. Here is a listing from the creation:

```sql
SQL> connect / as sysdba;
Connected.
SQL> @$ORACLE_HOME/rdbms/admin/dbmsol
```

Also, note that the `catproc.sql` script builds this using the following calls:

```sql
@catol.sql
@dbmsol.sql
@prvtol.plb
```

rem - If catol.sql and prvtol.sql aren't run you won't get the catalog views or the package body.

Note that this script creates the following synonyms and grants after installing the `sys.outln_pkg` package:

```sql
CREATE PUBLIC SYNONYM outln_pkg FOR sys.outln_pkg;
GRANT EXECUTE ON outln_pkg TO dba;
GRANT EXECUTE ON outln_pkg TO outln;

CREATE PUBLIC SYNONYM outline FOR sys.outln_pkg;
GRANT EXECUTE ON outline TO dba;
GRANT EXECUTE ON outline TO outln;
```

As we can see, you can reference the `outline` package by the name `outln_pkg` or `outline`. Now that the `outline` package is installed, you can now describe the `outline` procedure and see all of the available functions:

```sql
SQL> desc outline;
```

PROCEDURE DROP_BY_CAT
Next let's look at how to create a tablespace for the `outln` user to hold the stored outlines.

**Create the Stored Outline Tablespace**

The `outln` user requires a separate tablespace because your database may store thousands of stored outlines and it is a good idea to segregate the outlines into a separate tablespace for improved manageability. Here is the syntax to create the tablespace:

```sql
create tablespace outline_ts
  datafile '/u01/oradata/PROD/outline.dbf'
  size 10M
default storage
    (initial 10K next 20K minextents 1 maxextents unlimited)
online;
```

Tablespace created.

Next, we alter the `outln` user to use this tablespace by default.

```sql
alter user outln
default tablespace outline_ts;
```

User altered.

We are now ready to start capturing stored outlines. Let’s take a look at how it works.
How to Create and Modify a Stored Outline

The best way to show how to use stored outlines to change the execution plan for a SQL statement is to illustrate the procedure with a simple example. Let's start with a simple query to display the sum of all salaries for each department.

```sql
select
dname,
loc,
sum(sal)
from
emp,
depth
where
emp.deptno(+) = dept.deptno
and
depth.deptno = 10
group by
dname,
loc
;
```

Here is the output from this query:

<table>
<thead>
<tr>
<th>DNAME</th>
<th>LOC</th>
<th>SUM(SAL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
<td>8750</td>
</tr>
</tbody>
</table>

Find the Fastest Execution Plan

Now let's take a look at the execution plan for this query. Note that it uses `all_rows` optimization and does a full-table scan even though there is an index on the `emp` table:

```
OPERATION

OPTIONS                        OBJECT_NAME                    POSITION
------------------------------ ---------------------------- ----------
SELECT STATEMENT
SORT
GROUP BY
NESTED LOOPS
OUTER
TABLE ACCESS
```
Of course, this query will run faster if we direct the SQL to use the emp index to access the emp rows. When we reexecute the query with a rule hint, we see the full-table scan disappear.

```
select /*+ rule */
dname,
loc,
sum(sal)
from
emp,
depth
depth
where
emp.deptno(+) = dept.deptno
and
depth.deptno = 10
group by
dname,
loc
;
```

Let’s assume that we have done a timing of this query, and it is faster with rule-based optimization because it uses the dept_dept index instead of a full-table scan.

If we assume that the emp and dept tables are very large, then changing the SQL to use the index will improve the performance of the query. Of course, you should
always verify your hint by reexecuting the SQL with the SQL*Plus set timing on command to ensure that the hint improves performance. Now let’s create our first stored outline.

Create the Stored Outline for the Original Query
Outlines are created using the CREATE OUTLINE command. The syntax for this command is:

```
CREATE [OR REPLACE] OUTLINE outline_name
[FOR CATEGORY category_name]
ON sql_statement;
```

Where:

- **outline_name**  This is a unique name for the outline. Automatic outlines are stored as SYS_OUTLINE-nnn, where nnn is a large unique number.
- **[FOR CATEGORY category_name]**  This optional clause allows more than one outline to be associated with a single query by specifying multiple categories, each named uniquely.
- **ON sql_statement**  This specifies the SQL statement for which the outline is prepared.

To illustrate the syntax, here we create a stored outline for the original SQL.

```
create or replace outline cbo_sql
on
select
dname,
loc,
sum(sal)
from
emp,
depth
where
emp.deptno(+) = dept.deptno
and
dept.deptno = 10
group by
dname,
loc;
```
The query to display stored outline plans is very different in format than the explain plan utility. Here is the code to display the execution plan inside the $ol$hints$ table.

**outline.sql**

```sql
set echo off;
set verify off;
set feedback off;

column hint format a40;

select
    sql_text
from
    dba_outlines
where
    name = upper('&1');

select distinct
    lpad(' ',2*(level-1))||hint_text hint,
    hint#,
    table_tin,
    stage#
from
    outln.$ol$hints
start with
    hint#=1
connect by prior
    hint# = hint#-1
and
    ol_name = upper('&1')
order by
    stage#,
    hint#;
```

Now when we query the stored outline, we see a very different form of the execution plan:

```sql
SQL> @outline cbo_sql
SQL_TEXT
select
dname,
loc,
sum(sal)
from
emp,
dept
where
emp.deptno(+)

<table>
<thead>
<tr>
<th>HINT</th>
<th>HINT#</th>
<th>TABLE_TIN</th>
<th>STAGE#</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_EXPAND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO_FACT(EMP)</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>NO_FACT(DEPT)</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>FULL(EMP)</td>
<td>7</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>FULL(DEPT)</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Create a Stored Outline for the Hinted Query

In the next step, we create a new execution plan for the modified SQL statement. After creating this stored outline, we can then swap the stored outlines, replacing the original execution plan with our hinted execution plan.

```
create or replace outline
  rbo_sql
on
select /*+ rule */
  dname,
  loc,
  sum(sal)
from
  emp,
dept
where
  emp.deptno (+) = dept.deptno
and
  dept.deptno = 10
group by
dname,
loc
;```
Now, let's run outline.sql and see the new execution plan:

```
SQL> @outline rbo_sql
```

```sql
select /*+ rule */
  dname,
  loc,
  sum(sal)
from
  emp,
  dept
where
  emp.deptno(+) = dept.deptno
and
  dept.deptno = 10
group by
  dname,
  loc
```

<table>
<thead>
<tr>
<th>HINT</th>
<th>HINT#</th>
<th>TABLE_TIN</th>
<th>STAGE#</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOWRITE</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RULE</td>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>NOWRITE</td>
<td>8</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>NO_EXPAND</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>USE_NL(EMP)</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ORDERED</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>NO_FACT(EMP)</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>NO_FACT(DEPT)</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>INDEX(EMP DEPT_DEPT)</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>FULL(DEPT)</td>
<td>7</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Here we see that we have the stored outline for the original query and another stored outline for our tuned query. We are now ready to swap the stored outlines for these two queries so that the original query will use the improved stored outline.

**Swap the Stored Outlines**

The final step in the tuning is to swap the desired rbo_sql outline with our original cbo_sql outline. We will do this by running the swap_outlines.sql script. It will prompt you for the name of the original stored outline and the name of the improved stored outline. It will then swap the outlines between the two statements, transferring our desired execution plan to the original SQL statement (Figure 13-2).
Chapter 13: Tuning with Optimizer Plan Stability

swapped_outlines.sql

```sql
set echo off;
set feedback off;

prompt Enter the old outline name:
accept old

prompt Enter the new outline name:
accept new

update
  outln.ol$hints
set
  ol_name=decode(ol_name,upper('&new'),
                upper('&old'),upper('&old'),upper('&new'))
where
  ol_name IN (upper('&new'),upper('&old'));
commit;

prompt Here is the new outline for &old
@outline &old@outline &old
```

Here is the listing from `swapped_outlines.sql`. Note that it displays the original SQL source, but with our improved execution plan:

```sql
SQL> @swap_outlines
Enter the old outline name: cbo_sql
Enter the new outline name: rbo_sql

Here is the new outline for cbo_sql
```
Now that we understand how stored outlines are used to improve query performance, and how to tune a query using stored outlines, let’s move to a higher level and examine how you can manage stored outlines for your whole database.

### Managing Stored Outlines

Because the stored outline feature is new with Oracle8i, there are only a few rudimentary views and procedures to aid in the management of SQL stored outlines. As Oracle continues to enhance the functionality of optimizer plan stability, these tools will grow more robust.

Oracle provides two dictionary structures to aid in stored outline management, the DBA_OUTLINES view and the ol$hints table. Oracle also provides the outline package to aid in the categorization and management of stored outlines.

### Using the Dictionary Views and Tables for Stored Outlines

Oracle provides several views to help display stored outlines, most notably the DBA_OUTLINES view and the ol$hints table. Together, these structures will tell you
everything you need to know about the status of stored outlines in your database. Here are the columns in the DBA_OUTLINES view:

```
SQL> desc dba_outlines;
Name                        Null?    Type
------------------------------ -------- ----------------
NAME                          VARCHAR2(30)
OWNER                         VARCHAR2(30)
CATEGORY                      VARCHAR2(30)
USED                          VARCHAR2(9)
TIMESTAMP                    DATE
VERSION                      VARCHAR2(64)
SQL_TEXT                      LONG
```

We also have a DBA_OUTLINE_HINTS view, but it does not contain as much useful information as ol$hints, and it is seldom used because it lacks detail about the sequence of the stored outline steps.

```
SQL> desc dba_outline_hints;
Name                        Null?    Type
------------------------------ -------- ----------------
NAME                         VARCHAR2(30)
OWNER                        VARCHAR2(30)
NODE                         NUMBER
STAGE                        NUMBER
JOIN_POS                     NUMBER
HINT                          VARCHAR2(512)
```

To see the stored execution plans for a SQL statement, you must reference the outln.ol$hints view. Note that we created a public synonym for ol$hints to make it easily accessible by all users:

```
SQL> create public synonym ol$hints for outln.ol$hints;
Synonym created.
```

```
SQL> desc ol$hints;
Name                        Null?    Type
------------------------------ -------- ----------------
OL_NAME                      VARCHAR2(30)
HINT#                        NUMBER
CATEGORY                     VARCHAR2(30)
HINT_TYPE                    NUMBER
HINT_TEXT                    VARCHAR2(512)
STAGE#                       NUMBER
NODE#                        NUMBER
TABLE_NAME                   VARCHAR2(30)
TABLE_TIN                    NUMBER
TABLE_POS                    NUMBER
```
Once a stored outline is used, the used column in DBA_OUTLINES will change. Here we execute the original query and check to see if the stored outline was used:

```
SQL> select * from dba_outlines where name = 'CBO_SQL';
NAME                           OWNER
------------------------------ ------------------------------
CATEGORY                       USED      TIMESTAMP
------------------------------ --------- ---------  
VERSION                       ---------------------------
SQL_TEXT                       ---------------------------
CBO_SQL                        OPS$ORACLE
DEFAULT                       USED        14-APR-01
8.1.6.1.0
```

Here we can run a query to see our automatically generated outlines, as evidenced by their names in the SYS_OUTLINE-nnn.

```
SQL> select * from dba_outlines
       2 where name like 'SYS_OUTLINE%';
NAME                           OWNER
------------------------------ ------------------------------
CATEGORY                       USED      TIMESTAMP
------------------------------ --------- ---------  
VERSION                       ---------------------------
SQL_TEXT                       ---------------------------
SYS_OUTLINE_0104142013050001   READER
DEFAULT                       UNUSED    14-APR-01
8.1.6.1.0
SELECT ATTRIBUTE,SCOPE,NUMERIC_VALUE,CHAR_VALUE,DATE_VALUE FROM SYSTEM.PRODUCT_P
(SYS_OUTLINE_0104142013050002   READER
DEFAULT                       UNUSED    14-APR-01
8.1.6.1.0
SELECT CHAR_VALUE FROM SYSTEM.PRODUCT_PRIVS WHERE   (UPPER('SQL*Plus') LIKE UPPE
Using the Outline Package

The outline package contains several procedures that can help us manage our stored outlines. These procedures allow us to drop, categorize, and manage all stored outlines. The outline package contains the following functions and stored procedures.

- **Procedures**  drop_collision, drop_extras, drop_unrefd_hints, drop_unused, update_by_cat
- **Functions**  drop_collision_expact, drop_extras_expact, drop_unrefd_hints_expact

Let’s begin by looking at a procedure to identify and drop unused stored outlines.

Identify and Drop Unused Stored Outlines

One of the problems with systems that generate SQL with literal values is that there will be a huge number of nonreusable SQL statements. In these cases, cursor_sharing=force should be set to rewrite the SQL to replace the literals with host variables, thereby making the SQL reusable.

For example, cursor_sharing will transform this statement, removing the literal and replacing it with a host variable:

```sql
select * from customer where cust_name = 'JONES';
```

into a reusable form:

```sql
select * from customer where cust_name = :var1;
```
Oracle provides a procedure for dropping stored outlines that are not reused. Let's begin by running a query to see if our database has SQL statements that have never been reused.

```sql
SQL> set long 1000
SQL> select * from dba_outlines where used='UNUSED';
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>OWNER</th>
<th>CATEGORY</th>
<th>USED</th>
<th>TIMESTAMP</th>
<th>VERSION</th>
<th>SQL_TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST_OUTLINE</td>
<td>SYSTEM</td>
<td>TEST</td>
<td>UNUSED</td>
<td>08-MAY-99</td>
<td>8.1.3.0.0</td>
<td>select a.table_name, b.tablespace_name, c.file_name from dba_tables a, dba_tablespaces b, dba_data_files c where a.tablespace_name = b.tablespace_name and b.tablespace_name = c.tablespace_name and c.file_id = (select min(d.file_id) from dba_data_files d where c.tablespace_name = d.tablespace_name)</td>
</tr>
</tbody>
</table>

Now, we are ready to drop any stored outlines that have not been used. We do this by running the `drop_unused` procedure in the `outline` package.

```sql
SQL> execute outline_pkg.drop_unused;
PL/SQL procedure successfully completed.
```

SQL> select * from dba_outlines where used='UNUSED';
no rows selected

Remember, there is no recovery for this procedure, so you should always make sure that you no longer want the unused stored outlines before running `drop_unused`.

**Managing Categories with Stored Outlines**

Oracle provides a method whereby specific categories of stored outlines may be created. The categories can be very useful when you want to segregate stored outlines for testing purposes or when you want to separate stored outlines. However, the stored outlines in each category will always be used when a SQL statement enters the SQL parser, regardless of the category name, so it is important to remember that you cannot use stored outline categories to segregate...
stored outlines from execution. Categories are only used for the purpose of grouping related stored outlines. Let’s take a quick look at these procedures in the stored outline category.

**The drop_by_cat Procedure**
The `drop_by_cat` procedure drops all outlines that belong to a specific category. The procedure `drop_by_cat` has one input variable, `cat`, a VARCHAR2 that corresponds to the name of the category you want to drop.

```
create or replace outline
cbo_sql
for category
my_test
on select
dname,
loc,
sum(sal)
from emp,
depth
where emp.deptno(+) = dept.deptno
and dept.deptno = 10
group by
dname,
loc
;
```

Now, we can display all stored outlines in the `my_test` category:

```
SQL> set long 1000;
SQL> select * from dba_outlines where category='MY_TEST';
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>OWNER</th>
</tr>
</thead>
<tbody>
<tr>
<td>CATEGORY</td>
<td>USED</td>
</tr>
<tr>
<td>VERSION</td>
<td>SQL_TEXT</td>
</tr>
<tr>
<td>CBO_SQL</td>
<td>OPSSORACLE</td>
</tr>
<tr>
<td>MY_TEST</td>
<td>UNUSED</td>
</tr>
<tr>
<td>8.1.6.1.0</td>
<td></td>
</tr>
</tbody>
</table>
Oracle High-Performance SQL Tuning

The SQL query selects department name (dname), location (loc), and the sum of salaries (sum(sal)) from the employee (emp) and department (dept) tables. The query joins the two tables where the department number of the employee equals the department number of the department, and then groups the results by department name and location.

Now, to remove all stored outlines in the my_test category, we invoke the `drop_by_cat` procedure:

```
SQL> exec outline.drop_by_cat('MY_TEST');
PL/SQL procedure successfully completed.
SQL> select * from dba_outlines where category='MY_TEST';
no rows selected
```
Next, let’s look at the `update_by_cat` procedure.

The `update_by_cat` Procedure

The `update_by_cat` procedure merges all of the outlines in one category to a new category. This procedure is tricky because if a SQL in a stored outline already has an outline in the target category, then it is not merged into the new category. In other words, duplicates are not merged into the new category. Let’s illustrate this with a simple example.

Here we create three stored outlines. Please note that the SQL for `prod_sql1` is identical to the SQL in `test_sql1`.

```
create outline
    test_sql1
for category
test
on
    select * from dba_indexes;
create outline
test_sql2
```
for category
test
on
select * from dba_constraints;

create outline
prod_sql1
for category
prod
on
select * from dba_indexes;

Now we can select the names and categories and verify the placement of the SQL stored outlines in their categories:

```
SQL> select
    name,
category
from
dba_outlines
order by
category;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>CATEGORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROD_SQL1</td>
<td>PROD</td>
</tr>
<tr>
<td>TEST_SQL1</td>
<td>TEST</td>
</tr>
<tr>
<td>TEST_SQL2</td>
<td>TEST</td>
</tr>
</tbody>
</table>

Next, we execute the update_by_cat procedure to merge the TEST stored outlines into the PROD category.

```
SQL> exec outline.update_by_cat('TEST','PROD');
```

PL/SQL procedure successfully completed.

As we mentioned, because prod_sql1 is identical to test_sql1, we expect that the test_sql1 will not have been merged into the PROD category. Let's check and see:

```
SQL> select name,category from dba_outlines order by category;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>CATEGORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST_SQL2</td>
<td>PROD</td>
</tr>
<tr>
<td>PROD_SQL1</td>
<td>PROD</td>
</tr>
<tr>
<td>TEST_SQL1</td>
<td>TEST</td>
</tr>
</tbody>
</table>

Chapter 13: Tuning with Optimizer Plan Stability
To summarize, the update_by_cat is used to merge UNIQUE SQL statements from one stored outline category to another category. Duplicate stored outlines are not merged into the new category.

Conclusion

The optimizer plan stability feature of Oracle8i is an exciting new way to improve the speed of SQL and also provide a method for making tuning changes permanent. The major points of this chapter include these:

- In future releases of Oracle, stored outlines promise to improve SQL performance because SQL statements with stored outlines do not have to reformulate an execution plan when they are invoked.
- Stored outlines improve SQL tuning because an improved execution plan for a SQL statement can be stored without touching the original SQL source code.
- Stored outlines are great for tuning databases where the SQL source is not available, such as SAP and PeopleSoft applications products.
- Stored outlines can be turned on at the system level with the alter system set use_stored_outlines=true command. This will cause every SQL statement to store an outline with the name SYS_OUTLINE_nnn.
- Stored outlines can be set at the session level with the alter session set use_stored_outlines=true command. This is the recommended method for tuning individual SQL statements.
- You can change a stored outline for a SQL statement by creating and storing an improved stored outline and running the swap_outlines.sql script to swap in your improved stored outline.
- Databases with lots of nonreusable SQL with embedded literal values may quickly fill the outline tablespace unless cursor_sharing=force is set in the initialization file.
- Oracle provides a crude category management tool for creating and merging categories of stored outlines.

Next, let’s take a look at SQL tuning with the cost-based optimizer and see how you can maximize your benefits from exploiting the new cost-based hints.