Tipps & Tricks: Verbesserungen zum Thema Performance Tuning

Bereich:	DBA, Tuning	Erstellung:	02/2003 MP
Versionsinfo:	9.2.0.8, 10.2.0.4, 11.1.0.6	Letzte Überarbeitung:	07/2009 MM

Verbesserungen zum Thema Performance Tuning

Ab der Version 9i sind folgende Änderungen zu berücksichtigen

Neue Werte für den Cost-based Optimizer: FIRST_ROWS_n

Für den Initialisierungsparameter OPTIMIZER_MODE gibt es weitere Werte: FIRST_ROWS_n.

Bei dieser Einstellung versucht der Cost-Based Optimizer, alle SQL Statements so zu optimieren, dass die ersten n Zeilen der Abfrage so schnell wie möglich zurückgeliefert werden - n kann hierbei die Werte 1, 10, 100 und 1000 annehmen.

Beim entsprechenden neuen Hint FIRST_ROWS(n) - welcher direkt bei SQL Statements angegeben werden kann - können für n hingegen auch andere Werte verwendet werden.

Der bisherige Wert FIRST_ROWS kann auch weiterhin eingesetzt werden. Jedoch gilt es hier zu beachten, dass der Optimizer bei FIRST_ROWS neben den geschätzten Kosten (durch ANALYZE oder DBMS_STATS erzeugt) auch interne Regeln mit herangezogen hat, was bei Einsatz von FIRST_ROWS_n nicht mehr gemacht wird. Hier fließen nur mehr die geschätzten Kosten sowie der Wert für n ein. Bei hohen werden HASH JOINS, bei niedrigen Werten eher NESTED LOOP JOINS in Verbindung mit Indexzugriffen bevorzugt.

Der Parameter kann entweder in der SPFILE-Datei oder in der aktuellen Session gesetzt werden.

Beispiele:

```
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS_10;
```

```
SELECT /*+ FIRST_ROWS(1) */ *
   FROM HR.JOBS
 WHERE MIN_SALARY > 10000;
```

```
JOB_ID     JOB_TITLE                           MIN_SALARY MAX_SALARY
---------- ----------------------------------- ---------- ----------
AD_PRES    President                                 20000      40000
AD_VP      Administration Vice President            15000      30000
```

Neue Optimizer Hints

Es gibt noch weitere Hints, die jetzt bei SQL Abfragen eingesetzt werden können:

- NL_AJ: Verwendung von nested loop anti-joins bei einer NOT IN Unterabfrage
- NL_SJ: Verwendung von nested loop sort-joins bei einer EXISTS Unterabfrage
- CURSOR_SHARING_EXACT: siehe weiter unten
-

- FACT: Wird verwendet in Verbindung mit einer STAR Transformation und sagt aus, dass die betroffene Tabelle als Fakten-Tabelle verwendet werden soll.
- NO_FACT: Wird verwendet in Verbindung mit einer STAR Transformation und sagt aus, dass die betroffene Tabelle auf keinen Fall als Fakten-Tabelle verwendet werden soll.
- FIRST_ROWS (n): siehe weiter oben

Systemstatistiken

Neue Systemstatistiken erlauben es dem Cost-Based Optimizer (CBO) nun, neben I/O-Kosten auch CPU-Kosten bei der Berechnung der Ausführungspläne zu verwenden.

Bisher betrachtete der CBO nur die bei der Ausführung eines Statements geschätzten I/O-Kosten (über die Anzahl der errechneten Zugriffe auf Datenblöcke), nicht jedoch den u.a. bei Sortierungen notwendigen CPU Verbrauch.

Das Sammeln der Systemstatistiken geschieht immer über einen gewissen Zeitraum.

Während dieser Zeit sammelt Oracle Informationen über die aktuelle CPU und I/O Auslastung des Systems. Da diese Auslastung über einen ganzen Tag verteilt sehr unterschiedlich sein kann, empfiehlt es sich, Systemstatistiken über verschiedene Zeiträume anzufertigen, diese abzuspeichern und bei Bedarf zu aktivieren.

Hinweis:

Beim Sammeln der Systemstatistiken bleiben die bereits im Shared Pool liegenden und geparsten SQL Statements gültig. Das heißt, diese müssen - im Gegensatz zu einem ANALYZE auf Segmentebene - nicht neu geparsed werden. Leider bedeutet dies aber auch, dass die Ausführungspläne der bestehenden SQL Statements die neu gesammelten Statistiken noch nicht verwenden können.

Zum Erzeugen der Systemstatistiken gibt es neue Prozeduren im Package DBMS_STATS:

- GATHER_SYSTEM_STATS: Sammeln von CPU und I/O Statistiken
- GET_SYSTEM_STATS: Abrufen einzelner Systemstatistiken
- SET_SYSTEM_STATS: Setzen einzelner Systemstatistiken
- EXPORT_SYSTEM_STATS: Übertragen von Systemstatistiken aus dem Data Dictionary in eine Benutzertabelle
- IMPORT_SYSTEM_STATS: Übertragen von Systemstatistiken aus einer Benutzertabelle zurück in das Data Dictionary

Um Systemstatistiken erstellen zu können, benötigt ein Benutzer DBA Privilegien!

Hinweis:

Die meisten Prozeduren aus dem Package DBMS_STATS (auch oben genannte) führen zu Beginn und am Ende jeweils einen COMMIT durch.

Beispiel:

Wir gehen davon aus, dass ihr Datenbank System untertags von vielen Benutzern als Abfragesystem (OLTP) genutzt wird und nachts durch einige Batchläufe größere Datenmengen neu eingespielt werden. Diese doch sehr unterschiedlichen Systembelastungen sind gerade ein Paradebeispiel für den Einsatz der neuen Systemstatistiken.

Zu Anfang erzeugen wir die Tabelle TAB_SYSSTAT, in der die gesammelten Systemstatistiken gespeichert werden sollen:

```
BEGIN
  DBMS_STATS.CREATE_STAT_TABLE ( OWNNAME  => 'SYSTEM',
                                 STATTAB  => 'TAB_SYSSTAT',
                                 TBLSPACE => 'USERS' );
```

```
END;
/
```

Dann sammeln wir zuerst einmal Statistiken für den OLTP Betrieb von 09:00 bis 17:00:

```
BEGIN
  -- Start um 09:00
   DBMS_STATS.GATHER_SYSTEM_STATS ( INTERVAL => 480,
                                    STATTAB  => 'TAB_SYSSTAT',
                                    STATID   => 'OLTP' );
END;
/
```

Für den Batch-Betrieb sammeln wir Statistiken von 22:00 Uhr bis 05:00 Uhr.

```
BEGIN
  -- Start um 22:00
  DBMS_STATS.GATHER_SYSTEM_STATS ( INTERVAL => 420,
                                    STATTAB  => 'TAB_SYSSTAT',
                                    STATID   => 'BATCH' );
END;
/
```

Diese Statistiken können nun bei Bedarf geladen werden. Die OLTP Statistiken zum Beispiel immer morgens und die Batch Statistiken immer abends.

Beispiel für das Zurückspielen der OLTP Statistiken:

```
BEGIN
  -- START UM 08:30
  DBMS_STATS.IMPORT_SYSTEM_STATS ( STATTAB => 'TAB_SYSSTAT',
                                    STATID  => 'OLTP' );
END;
/
```

Hinweis:

Sie können den Prozess des regelmäßigen Austauschs der Statistiken mittels zweier Jobs auch automatisieren.

Identifizierung unnötiger Indizes (Index Überwachung)

Mit dieser Option kann man die Verwendung von Indizes überwachen. Nicht benutzte Indizes können über diesen Weg leichter gefunden und anschließend eliminiert werden.

Die Überwachung sollte über einen längeren und auch repräsentativen Zeitraum aktiviert werden, um dadurch möglichst alle den Index betreffenden SQL Statements mit überwacht zu haben.

```
-- Einschalten der Indexüberwachung
ALTER INDEX <indexname> MONITORING USAGE;
```

```
-- Ausschalten der Indexüberwachung
ALTER INDEX <indexname> NOMONITORING USAGE;
```

Die Verwendung eines Index kann über die View V$OBJECT_USAGE beobachtet werden.

Beispiel:

```
ALTER INDEX DEPT_ID_PK MONITORING USAGE;
```

```
SELECT * FROM V$OBJECT_USAGE;

INDEX_NAME TABLE_NAME  MON USE START_MONITORING    END_MONITORING
---------- ----------- --- --- ------------------- -------------------
DEPT_ID_PK DEPARTMENTS YES NO  03/06/2002 22:33:12
```

```
SELECT DEPARTMENT_ID FROM DEPARTMENTS WHERE DEPARTMENT_ID = 10;

DEPARTMENT_ID
-------------
           10
```

```
SELECT * FROM V$OBJECT_USAGE;

INDEX_NAME TABLE_NAME  MON USE START_MONITORING    END_MONITORING
---------- ----------- --- --- ------------------- -------------------
DEPT_ID_PK DEPARTMENTS YES YES 03/06/2002 22:33:12
```

```
ALTER INDEX DEPT_ID_PK NOMONITORING USAGE;
```

```
SELECT * FROM V$OBJECT_USAGE;

INDEX_NAME TABLE_NAME  MON USE START_MONITORING    END_MONITORING
---------- ----------- --- --- ------------------- -------------------
DEPT_ID_PK DEPARTMENTS NO  YES 03/06/2002 22:33:12 03/06/2002 22:34:21
```

In der Spalte USED sehen Sie, ob der Index seit Einschalten der Überwachung benutzt wurde (YES / NO).

Bei Verwendung von MONITORING USAGE löscht Oracle die zu diesem Index gehörenden Statistiken und sammelt neue, bis das Monitoring über die Option NOMONITORING USAGE wieder beendet wird. In der View V$OBJECT_USAGE kann neben der Verwendung des Index auch abgelesen werden, wann die Übwachung gestartet und gestoppt wurde.

Achtung:

Sollten Sie feststellen, dass ein Index im Überwachungszeitraum nicht benötigt wurde, so sollten Sie ihn nicht einfach löschen, sondern sich auf jeden Fall noch einmal vergewissern. Denn eventuell wurden doch nicht alle SQL Statements überwacht oder aber der Index dient nur zur Überprüfung einer Eindeutigkeit (PRIMARY KEY oder UNIQUE KEY) oder als Referenz für FOREIGN KEY Beziehungen.

Cursor Sharing: SIMILAR

SQL Statements müssen exakt identisch sein, um von Oracle wiederverwendet werden zu können. Ist dies nicht der Fall, weil z.B. der Vergleichswert in der WHERE-Bedingung immer wieder unterschiedlich ist, so muss Oracle jedes dieser Statements neu parsen und auch einen neuen Ausführungsplan erstellen. Um dies zu vermeiden, haben Sie zwei Möglichkeiten.

(1) Einsatz von Bind Variablen

Bei Verwendung von Bind Variablen sucht der Oracle Optimizer immer zuerst im Shared Pool nach bereits existierenden und verwendbaren SQL Statements.

Hinweis:

Der Ausführungsplan des im Shared Pool liegenden Statements wird unter Verwendung der Werte des ersten SQL Statements berechnet. Auch Statements mit anderen Werten für die Bind Variablen verwenden diesen Ausführungsplan!

(2) Einsatz des Parameters CURSOR_SHARING

Der Parameter kann die folgenden Werte annehmen:

- EXACT (Standardeinstellung: Keine Bind Variablen einsetzen)
- FORCE
- SIMILAR (Neu in 9i)

Wenn Sie den Parameter CURSOR_SHARING einsetzen, so versucht Oracle alle Literale eines Statements intern durch Bind Variablen zu ersetzen, um so zu erreichen, dass ähnliche Statements, die sich also nur im Wert der WHERE-Bedingung unterscheiden, identisch sind.

Beispiel:

Das folgende Statement

```
SELECT *  FROM hr.jobs WHERE min_salary > 10000;
```

wird intern zu (zu sehen in der View V$SQL):

```
SELECT * FROM hr.jobs WHERE min_salary > :"SYS_B_0"
```

Der Unterschied zwischen SIMILAR und FORCE ist, dass der Optimizer bei der Einstellung SIMILAR auch die eigentliche Verteilung der Werte innerhalb der Tabellen beachtet.

Kann er z.B. mittels Histogrammen erkennen, dass der bereits bestehende Ausführungsplan zu unnötigen Zugriffen führen würde, so würde dieser nicht verwendet werden.

Oracle würde in diesem Fall das Statement neu parsen und einen neuen Ausführungsplan erzeugen.

Die Angabe von FORCE würde den Optimizer jedoch trotzdem dazu zwingen, den im Shared Pool vorliegenden Ausführungsplan zu verwenden.

Gerade das ist der Grund, weshalb es in Data Warehouse Umgebungen nicht sinnvoll ist, CURSOR_SHARING auf FORCE zu setzen.

Sie können den Parameter CURSOR_SHARING im SPFILE ändern oder aber auch im laufenden Betrieb mittels ALTER SYSTEM, ALTER SESSION oder dem neuen Hint CURSOR_SHARING_EXACT:

Beispiel:

```
ALTER SESSION SET CURSOR_SHARING = SIMILAR;
```

Achtung:

Die Einstellung SIMILAR kann bislang (Stand: 11g) leider noch nicht uneingeschränkt empfohlen werden, da einige MetaLink-Dokumente diese als problematisch (buggy!) bezeichnen.

Index Skip Scannning

Zusammengesetzte (mehrspaltige) Indizes konnten bei Oracle Versionen vor 9i nur angesprochen werden, falls in der WHERE-Klausel auch die erste Spalte als Einschränkungskriterium verwendet wurde. Ab Oracle 9i kann der Optimizer einen Index Skip durchführen, um ROWID´s auch von nicht führenden Spalten zu erhalten.

Der Optimizer verwendet Statistiken, um zu entscheiden ob ein Skip Scan oder ein Full Table Scan (FTS) besser ist. Damit wird die Anzahl der benötigten Indizes für eine Tabelle und damit der Wartungsaufwand reduziert. Auch kann durch die Benutzung von wenigen sinnvollen Indizes viel Speicher eingespart werden.

Einige neue Spalten in der Tabelle PLAN_TABLE

In der Tabelle PLAN_TABLE gibt es neue Spalten, die u.a. die vom Cost-Based Optimizer geschätzten CPU und I/O Kosten beinhalten.

- CPU_COST: Ist proportional zur Anzahl der CPU Taktzyklen, die für die Ausführung des Statements geschätzt werden. Bei Verwendung des RBO ist dieser Wert null.
- IO_COST: Ist proportional zur Anzahl der zu lesenden Datenbankblöcke, die für die Ausführung des Statements geschätzt werden. Bei Verwendung des RBO ist dieser Wert null.
- TEMP_SPACE: Größe des temporären Speichers, der für die Bearbeitung des Statements benötigt wird (in Bytes). Bei Verwendung des RBO oder wenn kein temporärer Speicher benötigt wird ist dieser Wert null.

Mit dem folgenden SQL Statement können Sie eine Oracle 9i kompatible PLAN_TABLE-Tabelle erzeugen (oder über das Skript utlxplan.sql aus <ORACLE_HOME>\rdbms\admin anlegbar).

Beispiel:

```
CREATE TABLE PLAN_TABLE (
   STATEMENT_ID    VARCHAR2(30),
   TIMESTAMP       DATE,
   REMARKS         VARCHAR2(80),
   OPERATION       VARCHAR2(30),
   OPTIONS         VARCHAR2(255),
   OBJECT_NODE     VARCHAR2(128),
   OBJECT_OWNER    VARCHAR2(30),
   OBJECT_NAME     VARCHAR2(30),
   OBJECT_INSTANCE NUMERIC,
   OBJECT_TYPE     VARCHAR2(30),
   OPTIMIZER       VARCHAR2(255),
   SEARCH_COLUMNS  NUMBER,
   ID              NUMERIC,
   PARENT_ID       NUMERIC,
   POSITION        NUMERIC,
   COST            NUMERIC,
   CARDINALITY     NUMERIC,
   BYTES           NUMERIC,
   OTHER_TAG       VARCHAR2(255),
   PARTITION_START VARCHAR2(255),
   PARTITION_STOP  VARCHAR2(255),
   PARTITION_ID    NUMERIC,
   OTHER           LONG,
   DISTRIBUTION    VARCHAR2(30),
   CPU_COST        NUMERIC,
   IO_COST         NUMERIC,
   TEMP_SPACE      NUMERIC);
```

Und dasselbe für die Versionen 10/11g:

```
CREATE TABLE PLAN_TABLE (
   STATEMENT_ID       VARCHAR2(30),
   PLAN_ID            NUMBER,
   TIMESTAMP          DATE,
   REMARKS            VARCHAR2(4000),
   OPERATION          VARCHAR2(30),
   OPTIONS            VARCHAR2(255),
   OBJECT_NODE        VARCHAR2(128),
   OBJECT_OWNER       VARCHAR2(30),
   OBJECT_NAME        VARCHAR2(30),
   OBJECT_ALIAS       VARCHAR2(65),
   OBJECT_INSTANCE    NUMERIC,
   OBJECT_TYPE        VARCHAR2(30),
   OPTIMIZER          VARCHAR2(255),
   SEARCH_COLUMNS     NUMBER,
   ID                 NUMERIC,
   PARENT_ID          NUMERIC,
   DEPTH              NUMERIC,
   POSITION           NUMERIC,
   COST               NUMERIC,
   CARDINALITY        NUMERIC,
   BYTES              NUMERIC,
   OTHER_TAG          VARCHAR2(255),
   PARTITION_START    VARCHAR2(255),
   PARTITION_STOP     VARCHAR2(255),
   PARTITION_ID       NUMERIC,
   OTHER              LONG,
   DISTRIBUTION       VARCHAR2(30),
   CPU_COST           NUMERIC,
   IO_COST            NUMERIC,
   TEMP_SPACE         NUMERIC,
   ACCESS_PREDICATES  VARCHAR2(4000),
   FILTER_PREDICATES  VARCHAR2(4000),
   PROJECTION         VARCHAR2(4000),
   TIME               NUMERIC,
   QBLOCK_NAME        VARCHAR2(30),
   OTHER_XML          CLOB);
```

Gespeicherte Ausführungspläne

Ab Version 9i kann die View V$SQL_PLAN dazu benutzt werden, um die aktuellen Ausführungspläne der im Shared Pool befindlichen SQL-Statements zu sehen. Diese View enthält dieselben Informationen, wie die Tabelle PLAN_TABLE (EXPLAIN PLAN FOR ...).

Der Unterschied ist lediglich, dass über das Statement EXPLAIN PLAN FOR nur der theoretische Ausführungsplan gezeigt, wohingegen in V$SQL_PLAN der tatsächliche Ausführungsplan dokumentiert wird.

Eine noch recht umständliche Variante zur Ermittlung eines Ausführungsplans zeigt das folgende Beispiel:

```
COLUMN ID          FORMAT 999 NEWLINE
COLUMN OPERATION   FORMAT A20
```

```
COLUMN OPERATION   FORMAT A20
COLUMN OPTIONS     FORMAT A15
COLUMN OBJECT_NAME FORMAT A22 TRUNC
COLUMN OPTIMIZER   FORMAT A3  TRUNC

SELECT ID
     , LPAD (' ', DEPTH) || OPERATION OPERATION
     , OPTIONS
     , OBJECT_NAME
     , OPTIMIZER
     , COST
  FROM V$SQL_PLAN, V$SESSION
 WHERE HASH_VALUE = SQL_HASH_VALUE
   AND ADDRESS = SQL_ADDRESS
   AND SID = 9
 START WITH ID = 0
CONNECT BY
      (     PRIOR ID            = PARENT_ID
        AND PRIOR HASH_VALUE    = HASH_VALUE
        AND PRIOR CHILD_NUMBER = CHILD_NUMBER
      )
 ORDER SIBLINGS BY ID, POSITION
/
```

```
ID OPERATION            OPTIONS         OBJECT_NAME     OPT       COST
-- -------------------- --------------- --------------- --- ----------
 0 SELECT STATEMENT                                      CHO
 1  FILTER
 2   CONNECT BY         WITHOUT FILTERI
 3    COUNT
 4     NESTED LOOPS     OUTER
 5      NESTED LOOPS    OUTER
 6       MERGE JOIN
 7        SORT           JOIN
 8         FIXED TABLE  FULL            X$KSUSE
 9        SORT           JOIN
10         FIXED TABLE  FULL            X$KQLFXPL
11       TABLE ACCESS   BY INDEX ROWID  OBJ$
12        INDEX          UNIQUE SCAN     I_OBJ1
13      TABLE ACCESS    CLUSTER         USER$
14       INDEX           UNIQUE SCAN     I_USER#
```

Ab Version 10g ermitteln Sie sich nur noch die SQL_ID Ihres gewünschten Statements über die View V$SQLAREA und lassen sich danach einfach den formatierten Ausführungsplan ausgegeben über:

```
SELECT * FROM table(DBMS_XPLAN.DISPLAY_CURSOR('<sql_id'));
```

Ändern von Stored Outlines

Stored Outlines dienen der Stabilität von Ausführungsplänen. Ausführungspläne sind dadurch nicht mehr abhängig von der aktuellen Einstellung von Datenbankparametern oder vorhandenen Tabellen- oder Index-Statistiken, sondern nur mehr von den gespeicherten Outlines.

Die Outlines an sich bestehen aus einer Reihe von Hints, mit deren Hilfe der Cost-Based Optimizer einen Ausführungsplan erstellen kann.

Ab Oracle 9i ist es nun auch möglich, gespeicherte Outlines zu verändern und dadurch den eigenen Bedürfnissen noch besser anzupassen. Änderungen finden immer im privaten (lokalen) Bereich des jeweiligen Benutzers statt.

Um die geänderten (und hoffentlich auch getesteten) Stored Outlines wieder global zur Verfügung zu stellen, müssen sie explizit in den globalen Bereich zurückkopiert (veröffentlicht) werden.

Die folgenden Punkte können an einer Stored Outline geändert werden:

- Reihenfolge der Join Bearbeitung
- Art des Joins (NESTED LOOP, SORT MERGE, ...)
- Art des Tabellenzugriffs (FULL TABLE SCAN, INDEX RANGE SCAN, ...)

Lokale Änderungen können auf verschiedenen Wegen durchgeführt werden:

- über DML Operationen auf der Tabelle OUTLN.OL$HINTS
- über das Package DBMS_OUTLN_EDIT (SYS.OUTLN_EDIT_PKG)
- über den Outline Editor des Oracle Enterprise Manager Tuning Pack (Java Variante)

Voraussetzung zum Ändern an Outlines sind einige weitreichende Berechtigungen sowie folgende lokale Outline Tabellen (diese können über die Prozedur DBMS_OUTLN_EDIT.CREATE_EDIT_TABLES angelegt werden):

- SELECT_CATALOG_ROLE
- CREATE ANY OUTLINE, wenn Outlines erstellt werden sollen
- SELECT Rechte auf alle Tabellen, die in der FROM Klausel des CREATE OUTLINE Befehles referenziert werden
- EXECUTE ON DBMS_OUTLN_EDIT zum Erstellen und Editieren von privaten Outlines
- Tabelle OL$
- Tabelle OL$HINTS
- Tabelle OL$NODES

Anhand eines einfachen Beispiels wird das gesamte Vorgehen noch einmal praxisnah erläutert.

Beispiel:

Anlegen der lokalen Outline Tabellen

```
EXECUTE DBMS_OUTLN_EDIT.CREATE_EDIT_TABLES
```

Bestimmen der zu ändernden globalen Outline

```
SELECT name, sql_text
  FROM dba_outlines;

NAME                            SQL_TEXT
------------------------------- ------------------------------------
SYS_OUTLINE_011210201206740     SELECT * FROM SCOTT.DEPT
```

Bevor Änderungen an Stored Outlines durchgeführt werden können, müssen private Outlines aktiviert werden

```
ALTER SESSION SET use_private_outlines = true;
```

Erzeugen einer lokalen Kopie der globalen Outline

```
CREATE PRIVATE OUTLINE MY_1ST_OUTLINE
   FROM SYS_OUTLINE_011210201206740;
```

```
SELECT HINT#, HINT_TEXT
           FROM   OL$HINTS
          WHERE  OL_NAME = 'MY_1ST_OUTLINE'
          ORDER  BY HINT#;

HINT# HINT_TEXT
----- ----------------------------------------
    1 NOREWRITE
    2 RULE
    3 NOREWRITE
    4 NO_EXPAND
    5 ORDERED
    6 NO_FACT(DEPT)
    7 FULL(DEPT)
```

Durchführen einer Änderung an der lokalen Kopie mittels DML Statement

```
UPDATE ol$hints
   SET hint_text = 'EXPAND'
 WHERE hint# = 4;
```

Aktiviert wird die Änderung erst durch einen Refresh (alle bestehenden und von dieser Outline betroffenen Cursoren werden INVALID gesetzt und müssen dadurch neu übersetzt werden)

```
CREATE OR REPLACE PRIVATE OUTLINE MY_1ST_OUTLINE
   FROM PRIVATE MY_1ST_OUTLINE;
```

```
EXECUTE DBMS_OUTLN_EDIT.REFRESH_PRIVATE_OUTLINE ('MY_1ST_OUTLINE')
```

Zurückspielen der lokalen Kopie in den globalen Bereich

```
CREATE OR REPLACE OUTLINE SYS_OUTLINE_011210201206740
  FROM PUBLIC MY_1ST_OUTLINE;
```

Am Ende sollte man wieder die Verwendung globaler Outlines aktivieren:

```
ALTER SESSION SET use_private_outlines = false;
```

```
SELECT HINT_TEXT FROM OUTLN.OL$HINTS;

HINT_TEXT
----------------------------------------
NOREWRITE
RULE
NOREWRITE
EXPAND
ORDERED
```

```
NO_FACT(DEPT)
FULL(DEPT)
```

Fertig - ab jetzt steht die Änderung auch allen anderen Anwendern zur Verfügung!

Hinweis:

Auch wenn in Version 11g Outlines noch zur Verfügung stehen, wird dringend empfohlen auf das neue Feature "SQL Plan Management" umzusteigen. In zukünftigen Versionen sollen Outlines in der alten Form wegfallen.