Wenn sich der Cost Based Optimizer (CBO) für einen neuen Ausführungsplan entscheidet, ist das nicht immer zum Guten. Grund genug für den DBA, sich über das 11*g*-Feature „SQL Plan Management" zu informieren, mit dem sich Ausführungspläne stabilisieren lassen. Der Artikel zeigt dessen Praxistauglichkeit.

SQL Plan Management unter der Lupe

Jan Krüger, TUI Infotec GmbH

Wechselt der CBO plötzlich von einem „table access by index rowid" zu einem „full table scan direct path", drückt der neue Plan ein leistungsfähiges IO-Subsystem auch mal an die Wand: Produktionsausfall ist die Folge. Bei Änderungen in den Statistiken, „Bind variable Peeking" oder das 11*g*-Feature „Adaptive Cursor Sharing" kann niemand vorhersagen, wann ein Plan kippt.

Beim Einsatz von SQL Plan Management (SPM) speichert Oracle ausgewählte Ausführungspläne in Form von Optimizer Hints in der SQL Management Base im „SYSAUX Tablespace" – sie werden „Plan Baselines" genannt. Wird ein Statement geparst, für das ein oder mehrere Pläne in der Management Base hinterlegt sind, kommen andere Pläne für das Statement nicht in Frage. Soweit so einfach. Doch wie funktioniert das alles im Detail? Um Statements in die SQL Management Base als Plan Baselines aufzunehmen, gibt es im Wesentlichen zwei Möglichkeiten:

- Automatisches Aufzeichnen von Statements mithilfe des Parameters „OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES"
- Übernahme von Ausführungsplänen aus dem Cursor-Cache im Speicher der Instanz

```
set serveroutput on
variable n number;
declare
sql_rec v$sqlarea%rowtype;
i number := 0;
r number;
begin
for sql_rec in (select sql_id from v$sqlarea a
          where object_status='VALID'
          and parsing_schema_name IN ('SCHEMA')
          and parsing_user_id != 0
          and plan_hash_value != 0
        and exact_matching_signature != 0
          and executions > 99 and not exists
          (select 1 from dba_sql_plan_baselines b
           where b.signature = a.exact_matching_signature))
loop
r := DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE(sql_id=>sql_rec.sql_id);
DBMS_LOCK.SLEEP(1);
if (r = 0) then
dbms_output.put_line('no plan fixed from sql_id ' || sql_rec.sql_id);
end if;
if (r > 1) then
dbms_output.put_line(r || ' plans fixed from sql_id ' || sql_rec.sql_id);
end if;
i := i + r;
end loop;
:n := i;
end;
/
print :n;
```

Listing 1

Mit der Gießkanne

Setzt man den Parameter „OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES=true" im Systemkontext, werden im Folgenden alle Statements, die mehr als einmal ausgeführt werden, mit ihren Ausführungsplänen in der Plan Baseline verankert. Hinweis: Ab 11.2.0.3 gehört der Parameter zum Optimizer Environment, ein Umschalten erzeugt also einen Parse-Storm im System, weil die vorhandenen Cursors wegen Environment Mismatch invalidiert werden. Ob in einem Statement Bind-Variablen verwendet werden oder Literals, spielt für die Aufnahme in die Plan Baseline keine Rolle – zweimaliges Ausführen genügt. Es stehen hierbei keine Filterkriterien wie der User-Name zur Verfügung. Werden SQL-Statements von Überwachungs-Tools wiederholt ausgeführt, sind sie in der Baseline zu finden. Wer sich an einem solchen Overhead nicht stört, kann das Automatic-Plan-Capture auch dauerhaft anschalten und damit alle zukünftig auftretenden neuen Statements automatisch erfassen lassen. Hinweis: In 11.2.0.2 existiert ein Performance-Bug, der im Zusammenhang mit dem „Automatic Capture" auftritt. Der Parameter lässt sich auch im Session „Context" setzen, siehe den Anwendungsfall weiter unten.

Wer bei der Befüllung der Plan Baseline etwas gezielter vorgehen möchte, kann mithilfe der Funktion „DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE" Pläne aus dem Cursor-Cache

übernehmen. Die Funktion bietet einige vordefinierte Filter (nach „sqltext“, „parsing_schema_name“, „module“ sowie „action“) und kann auch mit der SQL_ID aufgerufen werden. Sollen nur einzelne Statements, die bereits durch kippende Pläne unangenehm aufgefallen sind, stabilisiert werden, ist der „load“ aus dem Cursor-Cache ebenso die Methode der Wahl. Die „sqlplus“-Ladeprozedur für den „bulk load“ erlaubt die freie Definition von Kriterien für die Übernahme (siehe Listing 1).

Der Select von „v$sqlarea“ legt fest, für welche SQL_IDs die Pläne in die Baseline übernommen werden sollen. Die Filterkriterien sind nach Bedarf anzupassen. Zur Beschleunigung werden hier die SQL_IDs ausgefiltert, die nicht für die Baseline in Frage kommen (etwa einfache INSERT-Statements ohne Plan), oder solche, die schon in der Baseline vorhanden sind. „SLEEP“ entzerrt den Parse-Storm, denn bei jedem neu in die Baseline aufgenommenen Plan wird der bisherige Cursor invalidiert, zu erkennen an „v$sql_shared_cursor.stb_object_mismatch“, STB steht für „SQL Tuning Base“, wozu auch die SPM-Baselines gehören. Schließlich hat die Übernahme aus dem Cursor-Cache den Vorteil, dass die dafür nötige Rechenzeit nicht in der User-Session anfällt, die das Statement ausführt, dessen Plan gerade in die Baseline aufgenommen werden soll.

Der Vollständigkeit halber sei noch darauf verwiesen, dass sich Pläne auch aus SQL-Tuning-Sets (DBMS_SPM.LOAD_PLANS_FROM_SQLSETS) und bei Migration aus älteren Oracle-Versionen aus „Stored Outlines“ (DBMS_SPM.MIGRATE_STORED_OUTLINES) in die Plan Management Baseline übernehmen lassen.

An der Basis

Die View „dba_sql_plan_baselines“ zeigt den Inhalt der Baseline an. „SIGNATURE“ repräsentiert den um Leerzeichen sowie Groß- und Kleinschreibung bereinigten SQL-Text; damit wird beim Parsen in der Baseline gesucht. „SQL_HANDLE“ besteht aus der Zeichenfolge „SQL_“ und der Signatur in Hexadezimal. „PLAN_NAME“ besteht aus der Zeichenfolge „SQL_PLAN_“, der Signatur in Base32-Kodierung (ab 11.2.0.3) und dem „plan hash value 2“ in Hexadezimal, den man an anderer Stelle auch unter der Bezeichnung „PLAN_ID“ wiederfindet.

Zu einem SQL-Statement kann es in der SQL Management Base mehrere mögliche Ausführungspläne geben. Jeder Plan hat fünf „YES/NO“-Flags (ab 11.2.0.2), die sich mithilfe der Funktion „DBMS_SPM.ALTER_SQL_PLAN_BASELINE“ vom DBA umschalten lassen.

Wird ein Plan erstmalig in die Baseline aufgenommen, sind die Flags auf „ENABLED=YES“ und „ACCEPTED=YES“ gesetzt. Alle nachfolgenden Pläne für das gleiche SQL sind „ENABLED=YES“, aber zunächst „ACCEPTED=NO“. Sie müssen erst freigegeben werden, bevor sie für die Ausführung in Frage kommen (siehe weiter unten).

„ENABLED (YES/NO)“ gibt an, ob der Plan überhaupt für die Ausführung oder eine Freigabe zur Ausführung in Betracht kommt. Setzt der DBA mithilfe von „DBMS_SPM.ALTER_SQL_PLAN_BASELINE“ das „ENABLED“-Flag auf „NO“, wird der Plan nicht mehr benutzt und ist auch kein Kandidat mehr für eine Freigabe. Würde der Plan stattdessen aus der Baseline entfernt – was mit „DBMS_SPM.DROP_SQL_PLAN_BASELINE“ möglich ist –, käme er wahrscheinlich nach kurzer Zeit wieder als Kandidat („ENABLED=YES“, „ACCEPTED=NO“) in die Baseline. Die anderen Flags bekommen wir später.

Das beliebte Paket „DBMS_XPLAN“ wurde dahingehend erweitert, dass die Funktion „display_cursor“ in den Anmerkungen darauf hinweist, wenn eine SPM-Baseline verwendet wird. Mit „dbms_xplan.display_sql_plan_baseline“ kann man den Plan, der in einer Baseline steckt, anzeigen, mit der „ADVANCED“-Option auch die zugrunde liegenden Outlines (zu den Einschränkungen weiter unten).

Der SQL-Text ist Dreh- und Angelpunkt

Ist die Plan Baseline erfolgreich gefüllt, das automatische Capture aus- und das Plan Management angeschaltet (default ist der Parameter „OPTIMIZER_USE_SQL_PLAN_BASELINES=true“), läuft für jedes zu parsende Statement etwas vereinfacht Folgendes ab:

1. Das Statement wird geparst und der optimale Ausführungsplan wird bestimmt, genauso wie ohne SQL Plan Management.
2. Mithilfe der „exact_matching_signature“, die den SQL-Text, bereinigt um Formatierung etc., repräsentiert, wird in der SQL Management Base nachgeschaut, ob eine Baseline vorhanden ist.
3. Existiert eine Baseline, wird geprüft, ob der soeben neu ermittelte Plan enthalten ist und benutzt werden darf („ENABLED=YES“, „ACCEPTED=YES“). Wenn ja, wird der Plan genutzt, ansonsten wird er zur Baseline hinzugefügt („ENABLED=YES“, „ACCEPTED=NO“) und das Statement wird mithilfe der in der Baseline hinterlegten Pläne neu geparst, um daraus den besten Ausführungsplan zu bestimmen.

Der genaue Ablauf unter Einbeziehung des „FIXED“-Flags ist im Flussdiagramm dargestellt. „FIXED=YES“ gibt den Plänen Priorität über die „FIXED=NO“-Pläne und verhindert außerdem, dass neue Plan-Kandidaten aufgenommen werden. Die Auswahl der Pläne über die „exact_matching_signature“ – also nur den SQL-Text – bedeutet, dass gleiche SQLs, die in verschiedenen Usern und damit verschiedenen Schemata ablaufen, die gleichen Pläne bekommen. Ob dies ein Bug oder Feature ist, muss jeder selbst entscheiden. In einer mandantenfähigen Applikation, in der für jeden Mandanten ein Schema mit den gleichen Objekten vorgesehen ist, kann dies zu Problemen führen, wenn die Datenmengen und Verteilungen unterschiedlich sind. Bei Userzugriff auf die Datenbank ist es sogar möglich, dass ein Benutzer dem anderen schlechte Ausführungspläne unterschiebt, indem er in seinen Schema-Baselines SQL-Statements erzeugt, die auch in anderen Schemata benutzt werden. Für viele Anwendungsfälle dürfte diese Einschränkung zwar keine Rolle spielen, ganz zu Ende gedacht ist das aber noch nicht.

Adaptive Cursor Sharing und Plan Management

In der Datenbank-Version 11*g* wurde die Ausführungsoptimierung für SQL-

```
SQL_ID  4ksb3mrktdvqp, child number 0
-------------------------------------
select count(iata_cd) from airport where country=:c
Plan hash value: 765729859
-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name        | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |      1 |        |      1 |00:00:00.02 |      59 |     59 |
|   1 |  SORT AGGREGATE               |             |      1 |      1 |      1 |00:00:00.02 |      59 |     59 |
|   2 |   TABLE ACCESS BY INDEX ROWID| AIRPORT     |      1 |   1323 |   1323 |00:00:00.02 |      59 |     59 |
|*  3 |    INDEX RANGE SCAN           | COUNTRY_IDX |      1 |   1323 |   1323 |00:00:00.01 |       6 |      6 |
-----------------------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("COUNTRY"=:C)
Note
-----
   - SQL plan baseline SQL_PLAN_7p565u74hm99b45fcffbd used for this statement
```

Listing 2: Ausgabe von „dbms_xplan.display_cursor“ mit Hinweis auf die verwendete Plan Baseline im Notes-Abschnitt

Statements mit Bind Variablen weiterentwickelt. Man stelle sich eine Tabelle mit den Flughäfen der Welt vor, neben dem Namen des Flughafens und der Stadt ist auch das Land enthalten, in dem der Flughafen liegt; auf der Länder-Spalte wurde ein Index angelegt (siehe Abbildung 1). Sollen nun beispielsweise die Flughäfen in den USA selektiert werden, ist ein Full Table Scan unter Umständen effektiver als ein Zugriff über den Index, weil es dort so viele Flughäfen gibt, dass die meisten Blöcke der Tabelle ohnehin inspiziert werden müssen. Selektiert man die Flughäfen eines Landes wie Gambia mit nur einem Flughafen, empfiehlt sich der Zugriff über den Index (siehe Listing 2 und Listing unter www.doag.org/go/doagnews/krueger_listing).

Wohingegen in der Version 10*g* nur beim erstmaligen Parsen der Wert der Bind-Variablen herangezogen wurde (bind variable peeking), überwacht 11*g* die Variablenwerte und Ausführungspläne dauerhaft mit dem Adaptive Cursor Sharing (ACS). Wie schon in 10*g* sagt der CBO mithilfe von Histogrammen die Selektivität auf Basis des Bind-Variablen-Wertes voraus. Neu ist, dass durch Cursor-Children zu einem SQL-Statement mehrere Pläne im Cursor-Cache vorgehalten werden,

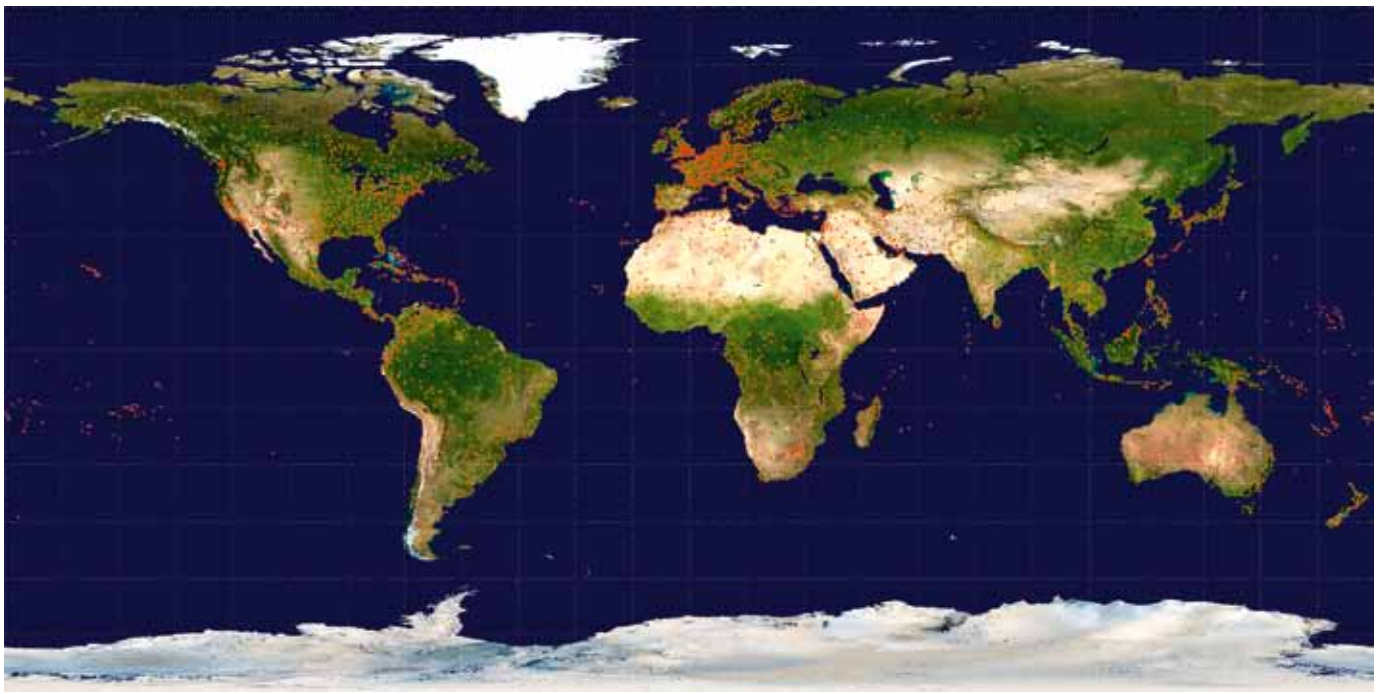

Abbildung 1: Beispiel-Datensatz „Flughäfen der Welt“. Der Banjul International Airport von Gambia liegt an der Küste von Westafrika. Gambia ist ein kleines Land, eingeschlossen vom Senegal (Karte und Daten: openflights.org)

die dann je nach Variablenwerten zur Ausführung angewählt werden. ACS ist hier nicht das Thema, deswegen sei der Ablauf nur kurz erklärt: Kommt ein SQL für ACS in Frage, wird dies in einem Flag vermerkt (siehe „v$sql.is_bind_sensitive“). Die folgenden Ausführungen werden hinsichtlich Größe der Ergebnismenge, CPU-Verbrauch und Consistent Reads stichprobenhaft überwacht („v$sql_cs_histogram“ und „v$sql_cs_statistics“). Mithilfe dieser Historie wird entschieden, dass das SQL von verschiedenen Plänen profitieren könnte. Sodann wird neu geparst und es entsteht ein „bind aware cursor“ („v$sql.is_bind_aware“). Für diesen werden die Bind-Variablen-Werte-Bereiche und Selektivitäts-Prognosen dokumentiert („v$sql_cs_selectivity“).

Liegt der Wertebereich einer neuen Ausführung außerhalb der bereits bekannten Bereiche, wird neu geparst. Kommt dabei ein Plan zustande, der bereits in einem anderen Child-Cursor vorliegt, werden die Cursor- und Werte-Bereiche zusammengefasst. Auf diese Weise entsteht nach und nach ein eingeschwungener Zustand mit einer begrenzten Zahl von „überlebenden“ Child-Cursors und es sind schließlich keine weiteren Parses mehr notwendig.

Allerdings hängt der tatsächliche Ablauf im ACS von der Reihenfolge des Auftretens verschiedener Bind-Variablen-Werte ab. In der Einschwung-Phase werden die Ausführungspläne nicht optimal angewandt, schlimmstenfalls geht es zu wie beim „bind variable peeking“ in älteren Versionen. Es ist damit letztlich nicht vorhersagbar, ob und wann ein Cursor „bind aware“ wird und wann ein eingeschwungener Zustand erreicht ist, bei dem die Pläne optimal genutzt werden. Zum genannten Beispiel: Es können viele Full Table Scans für Gambia ausgeführt werden, bevor der Bedarf für einen angepassten Plan erkannt wird. Übrigens, der „BIND_AWARE“-Hint hilft dem CBO bei Bedarf auf die Sprünge.

Funktioniert Plan Management also mit „Adaptive Cursor Sharing“? Im Prinzip ja, denn es kommt erst ins Spiel, nachdem der normale CBO-Lauf einschließlich der Optimierungen zur „bind awareness“ ausgeführt wurde. Er wirkt erst im Nachhinein als Filter und Korrektiv auf den Ergebnissen des CBO.

Hilft das SQL Plan Management bei den Problemen des „Adaptive Cursor Sharing“? Leider nein, denn es tritt erst in Erscheinung, nachdem der normale CBO-Lauf einschließlich der Optimierungen zur „bind awareness“ ausgeführt wurde. Ob ein Cursor „bind aware“ ist, entscheidet nicht das SPM (Stand 11.2.0.3). Die Baseline beinhaltet zwar idealerweise alle Pläne, die beim ACS herauskommen können und abhängig von den Variablen-Werten gute Performance bieten. Wann und ob der CBO den „cursor bind aware“ macht, beeinflusst das SPM nicht, auch speichert es nicht die Informationen zu Bind-Variablen-Werten und zur Selektivität. Diese müssen beispielsweise nach einem Restart jedes Mal neu erarbeitet werden. Insofern hat die Stabilisierung der Pläne durch SPM Grenzen: Was im ACS schiefgehen kann, bis ein eingeschwungener Zustand erreicht ist, geht auch mit SPM schief. Der Full Table Scan für Gambia bleibt uns also erhalten.

Die Zusammenarbeit zwischen SPM und ACS hält überdies noch eine unangenehme Überraschung bereit: Setzt man den automatischen Plan Capture per Parameter ein, um eine Baseline zu erzeugen, wird nur der erste Plan mit „ACCEPTED=YES“ gespeichert. Alle weiteren Pläne, die dem ACS entspringen, werden als Kandidaten mit „ACCEPTED=NO“ in der Baseline gespeichert und können erst mal nicht genutzt werden. Der Full Table Scan für Gambia, der ohne SPM irgendwann verschwinden würde, bleibt also dauerhaft erhalten, wenn man nicht eingreift. Wer glaubt, mit dem Setzen des Parameters sei alles getan, könnte hier ein böses Erwachen erleben.

Der Lauf der Welt

Applikationen – auch solche, die Datenbanken nutzen – entwickeln sich für gewöhnlich weiter. Im Plan Management kann einiges passieren, auf das reagiert werden muss:

- Die Datenbestände ändern sich und ein neuer Ausführungsplan wird (vermeintlich) effektiver als der im SPM festgehaltene
- Neue SQL-Statements kommen hinzu, für die keine Baseline existiert
- Alte SQL-Statements werden nicht mehr genutzt
- Ein Plan in der Baseline kann nicht mehr genutzt werden, weil sich im Datenmodell etwas geändert hat – zum Beispiel eine zusätzliche Spalte im Index
- Ein neuer Plan wäre besser, weil sich im Datenmodell etwas geändert hat

Werden Pläne in der Baseline länger nicht benutzt, kommt ein Auf-

```
select count(*) kandidaten
from dba_sql_plan_baselines
where enabled='YES' and ACCEPTED='NO'
```

Listing 3

```
select count(distinct exact_matching_signature) neues_sql
from v$sqlarea a
        where object_status='VALID'
        and parsing_schema_name IN (‚SCHEMA’)
        and parsing_user_id != 0
        and plan_hash_value != 0
        and exact_matching_signature != 0
        and executions > 99 and not exists
        (select 1 from dba_sql_plan_baselines b
        where b.signature = a.exact_matching_signature
        and b.reproduced=’YES’)
```

Listing 4

räumjob und löscht sie. „DBMS_SPM.CONFIGURE“ ist bei der Einstellung der Aufbewahrungsdauer behilflich. Neue Pläne für in der Baseline vorhandene SQL-Statements fügt Oracle automatisch hinzu („ENABLED=YES“, „ACCEPTED=NO“) und damit ist die Überwachung einfach (siehe Listing 3).

Neue SQL-Statements (ohne Pläne in der Baseline) sind da schon etwas kniffliger. Schaltet man das Auto-Capture dauerhaft an, stellt sich die Frage nicht. Geht man selektiver vor, empfiehlt sich im Monitoring ein Abgleich mit „v$sqlarea“ mit den gleichen Selektions-Kriterien, die auch für die Befüllung der Baseline genutzt werden sollen (siehe Listing 4).

Bleiben noch die Pläne, die nicht mehr funktionieren. Ab 11.2.0.2 gibt das Flag „dba_sql_plan_baselines.reproduced“ darüber Auskunft, ob es beim letzten Versuch geklappt hat, den Plan zu nutzen. Das SQL zur Überwachung ist dem Leser überlassen. Am Rande sei bemerkt, dass ein SQL-Statement, dessen Plan in der Baseline in einem Schema funktioniert und im anderen nicht, weil beispielsweise ein Index nur in einem Schema vorhanden ist, hier zu munterem Flag-Blinken führen kann.

Übrigens: Einen Plan, der wegen Änderungen im Datenmodell nicht mehr reproduziert werden kann, zeigt „dbms_xplan.display_sql_plan_baseline“ auch nicht mehr an. Stattdessen wird ein anderer Plan gemeldet, der alternativ zur Baseline zum Einsatz kommt. Ob das ein Bug oder Feature ist, mag jeder selbst bewerten, zumindest einen Hinweis in der „dbms_xplan“-Ausgabe wäre nützlich. Ebenso ist es nicht mehr möglich, mit „dbms_xplan“ Baselines anzuzeigen, wenn das als „CREATOR“ in der Baseline hinterlegte Schema nicht (mehr) vorhanden ist. Genutzt werden kann die Baseline trotzdem. Das ist keine Überraschung, denn die Baseline enthält ja nicht den tatsächlichen Plan, sondern Hints, um ihn herzustellen. Für den „explain“ benötigt „dbms_xplan“ ein Schema mit entsprechenden Objekten, sodass der SQL-Text zusammen mit den Hints auch einen Plan ergibt. Dennoch ist hier noch Potenzial für Verbesserungen.

```
variable n number
exec :n :=DBMS_SPM.ALTER_SQL_PLAN_BASELINE(NULL, 'SQL_PLAN_ccb833bb-
deca1852', 'enabled', 'NO');
```

Listing 5

```
exec :n :=DBMS_SPM.ALTER_SQL_PLAN_BASELINE(NULL, ,SQL_PLAN_ccb833bb-
deca1852', ,autopurge', ,NO');
```

Listing 6

Darwinismus unter Plänen

Hat unser Monitoring festgestellt, dass es neue Pläne gibt, die besser sein sollen als die alten, ist der DBA gefragt. Was soll er tun? Alle neuen Pläne mit „dbms_xplan“ ausgeben, begutachten und irgendwie erfühlen, ob sie besser sind als die alten, kommt nicht in Frage. „DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE“ ist der versprochene Heilsbringer, ein „verify=>‘YES‘“ bittet die Funktion, vor der Freigabe eines Plans („ACCEPTED=YES“) doch mal nachzusehen, ob er wirklich besser ist als der alte. Doch leider funktioniert nicht, was nicht funktionieren kann. Mit welcher Magie der Verify vorgeht, ist nicht vollständig dokumentiert; es sieht danach aus, als würden die Statements mit den Plänen und dem zu ihrer Entstehung gehörenden Satz Bind-Variablen-Werte tatsächlich ausgeführt und vermessen. Nur leider dauert ein Zugriff auf die Flughäfen der USA eben länger als auf den von Gambia, auch wenn der Full Table Scan schneller wäre als der Indexzugriff. Hat Gambia die Baseline gelegt, können die USA lange darauf warten, dass die Plan-Evolution mit „verify=>‘YES‘“ ihnen weiterhilft.

Will der DBA die Unschärfen des Verify im „EVOLVE“ vermeiden, bleibt also nur das „verify=‘NO‘“, was alle Pläne mit „ENBLE=YES“ in den „ACCEPTED=YES“-Status befördert. Ein Trost bleibt: Wann Pläne freigegeben werden, kann man planen und im Rahmen von Change-Prozessen steuern, sodass der mögliche Schaden geringer ist, als wenn die Pläne zur Unzeit gekippt wären. Ein zweiter Trost: Stellt sich ein neuer Plan nach der Freigabe als schädlich heraus, lässt er sich schnell und einfach abschalten, um zum alten Plan zurückzukehren:

1. Auf herkömmlichem Weg das SQL-Statement mit dem schlechten Plan ausfindig machen
2. Aus der Spalte „v$sql.sql_plan_baseline“ den Namen des schlechten Plans entnehmen, hier als Beispiel „SQL_PLAN_ccb833bbdeca1852“
3. Plan abschalten (siehe Listing 5)
4. Automatisches Löschen ausschalten (siehe Listing 6)

Im Zuge dessen werden automatisch die Cursors mit dem schlechten Plan invalidiert und es wird mit dem bisherigen Plan aus der Baseline, der ja immer noch da ist, weitergearbeitet. Alternativ kann man die anderen Pläne für das SQL auch auf „FIXED=YES“ umschalten. Verzichtet man auf die Fixierung des Plans und bleibt der „AUTOPURGE=YES“, wird der abgeschaltete Plan nach der Aufbewahrungszeit gelöscht (er wird ja nicht mehr benutzt), es entsteht ein neuer Kandidat, der irgendwann „evolved“ wird, und dann geht das Problem von vorne los.

Gut verpackt auf Reisen

Sicherlich erscheint es für Applikationshersteller reizvoll, eine Baseline mit der Applikation mitzuliefern, um Probleme mit Ausführungsplänen von vornherein auszuschließen. Eine andere Idee ist, Baselines, die sich in der Qualitätssicherung behauptet haben, in ein Produktionssystem zu übernehmen. Bei solchen Vorhaben helfen die Funktionen „DBMS_SPM.PACK_STGTAB_BASELINE“ und „DBMS_SPM.UNPACK_STGTAB_BASELINE“, die die

```
variable n number
exec :n := DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE(sql_
id=>'7vw4wyf41cd3a',  sql_handle=>'SQL_7a94c5d1c909a52b');
```

Listing 7

```
exec :n := DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE(sql_
id=>'7vw4wyf41cd3a',  sql_text=>'select count(*) from airport where
country=:c');
```

Listing 8

Baseline in eine Staging-Tabelle übertragen oder aus einer solchen in die SQL Management Base aufnehmen. Die Funktionen können bei Bedarf nach verschiedenen Kriterien filtern. Die Staging-Tabelle, die sich mit „export", „import" oder „datapump" von einem System ins andere transferieren lässt, erzeugt man zuvor mit „DBMS_SPM.CREATE_STGTAB_BASELINE". „UNPACK" überschreibt zum Beispiel die Flags vorhandener Pläne. Will man also im Rahmen eines neuen Software-Release Pläne hinzufügen, kann es sinnvoll sein, zunächst die vorhandene Baseline mit der zu importierenden zu vergleichen und das Delta für den „UNPACK" bereitzustellen.

Dem CBO auf die Sprünge helfen

Außer zur Stabilisierung kann das SPM auch genutzt werden, um Ausführungspläne zu ändern. Wenn das SQL in einer Applikation verborgen und nicht mit Hints versehen werden kann, ist dies eine elegante Möglichkeit neben dem SQL-Patch und der Manipulation von Statistiken. Zunächst benötigt man den Original-SQL-Text, der zum Beispiel aus „dba_sql_plan_baselines" oder aus „v$sql" extrahiert werden kann. Am einfachsten ist für den DBA, wenn es gelingt, durch Änderungen des „Optimizer Environment" im „session context" (beispielsweise über „alter session set optimizer_index_cost_adj=1") den gewünschten Ausführungsplan zu erhalten. Indem man „optimizer_capture_sql_plan_baseline" in der Session auf „true" setzt und das Statement zweimal ausführt, fügt man die gewünschte Baseline hinzu. Nun muss nur noch der gewünschte Plan auf „EVOLVE" gesetzt werden (verify=>'NO') und der unerwünschte auf „ENABLED=NO".

Etwas komplizierter ist der Vorgang, wenn man einen Hint im SQL-Text anbringen muss, um den gewünschten Plan zu bekommen. Der SQL-Text hat dann eine andere Signatur. Idealerweise ist das betreffende SQL-Statement für den Hint mit seinem schlechten Plan in der Baseline bereits enthalten. Aus „dba_sql_plan_baseline" bestimmt man den „SQL_HANDLE". Anschließend führt man das SQL-Statement mit dem Hint aus und bestimmt aus „v$sql" die SQL_ID sowie – falls nicht eindeutig – den „plan_hash_value". Schließlich wird der neue Plan aus dem Cursor-Cache zur Baseline hinzugefügt, wobei man das Zielstatement über „SQL_HANDLE" referenziert (siehe Listing 7).

Steht „SQL_HANDLE" nicht zur Verfügung, weil das Statement noch nicht in der Baseline enthalten ist, kann auch der SQL-Text an die Funktion übergeben werden; „SQL_HANDLE" wird dann aus dem Text errechnet (siehe Listing 8).

Was das Ganze kostet

Das Feature ist Bestandteil der Oracle-Enterprise-Edition-Lizenz und es sind keine zusätzlichen Optionen wie etwa Tuning Pack erforderlich, um es zu nutzen. Trotz der noch nicht ganz ausgestandenen Kinderkrankheiten sollte jeder DBA die Funktionalität für seinen Troubleshooting-Werkzeugkasten kennen und bei Bedarf zumindest für einzelne Statements einsetzen können.

Jan Krüger
jan.krueger@tui-infotec.com

Wir begrüßen unsere neuen Mitglieder

Persönliche Mitglieder

Thomas Ritter
Sybille Kopp
Viktor Schmidt
Klaus Speierl
Sascha Oberhollenzer
Oliver Mang
Matthias Schulz
Tammo Meyer
Matthias Deuß
Armin Eberle
Rene Renk
Waldema Schott
Andres Campo-Penuela
Jörn Kunnert
Kirill Babeyev
Katrin Marchewka
Gerold Geiß
Marian Schmöker
Andreas Hild
Rainer Horst
Silvio Nifkiffa
Andreas Oberacher
Rainer Ackermann
Kersten Bühnert
Norman Stöcker
Oliver Horlacher
Ralph Baumbach
Manuel Swiercz
Muthiah Chidambaram

Firmenmitglieder

Jan Ross, PPI Aktiengesellschaft
Susanne Herbst, Baur Versand GmbH & Co. KG
Dr. Renko Ungruhe, items GmbH
Ralph Baumbach, avato consulting ag
Marco Trujka, WERTGARANTIE Management GmbH
Christian Bretting, Georg-Simon-Ohm Hochschule