Wie kommt der Hint in das SQL, ohne die Anwendung zu ändern?

Mathias Zarick
Trivadis Delphi GmbH
Wien

Schlüsselworte
SQL Tuning, Hints, Plan Stability, Stored Outlines, SQL Profiles, SQL Patches, SQL Plan Baselines

Einleitung
Jeder, der sich intensiv mit der Oracle Datenbank und vor allem mit dem SQL Tuning beschäftigt, kennt Hints. Was sind Hints? Sie erlauben es, den Oracle Optimizer und vor allem seine Entscheidungen zu beeinflussen. Er folgt ihnen, wenn sie anwendbar sind. Sie werden innerhalb von Kommentaren im SQL Statement eingebracht. Sie müssen nach dem ersten Schlüsselwort pro Query Block erscheinen. Innerhalb des Kommentares muss das erste Zeichen ein + sein.
Es gibt 2 Kategorien von Hints: solche, die wirklich den Optimizer beeinflussen, wie zum Beispiel `FULL, INDEX, LEADING, ORDERED, …` und solche, die gewisse alternative Verhaltensweisen der Operation ermöglichen, wie zum Beispiel `APPEND, CACHE, MONITOR, GATHER_PLAN_STATISTICS, RESULT_CACHE ….`
Welche Hints gibt es? Schauen Sie in der View `v$sql_hint` nach!
Warum sollten wir Hints verwenden, oder eben nicht verwenden? Hints stellen häufig nur einen Workaround dar, um gewisse sub-optimale Ausführungspläne zu umgehen. Solche Workarounds sollten keine dauerhafte Lösung sein. Daten und Kardinalitäten können sich im Laufe der Zeit ändern, daher ist es wichtig, dem Optimizer eine Chance zu geben, auf solche Änderungen zu reagieren. Außerdem kann man davon ausgehen, dass der Optimizer von Release zu Release etwas cleverer wird und gewisse Altlasten sicher obsolet sind. Ferner können Hints in gewissen Testszenarien hilfreich sein, um „was wäre wenn“-Analysen durchzuführen, vor allem, wenn das Testsystem keine ausreichenden Datenbestände vorhält. Einige Hints müssen aber bewusst gesetzt werden, um ein gewisses Verhalten des Statements zu erzwingen. Ein Beispiel dafür wäre der `APPEND` Hint, es gibt keine andere Möglichkeit einen direct path load zu erreichen, als diesen Hint zu platzieren. Andere Beispiele dafür sind `BIND_AWARE` oder `RESULT_CACHE`.
Jonathan Lewis‘ Regeln für das Setzen von Hints besagen in erster Linie: Tun Sie es nicht! Falls Sie es tun müssen, so gehen Sie davon aus, dass Sie irgendwo einen Fehler gemacht haben. Im Abspann dieses Mantras gesteht er aber auch ein, dass man Sie in manchen Fällen brauchen wird.
In diesem Vortrag soll es nun nicht weiter darum gehen, ob Hints sinnvoll oder nicht sinnvoll sind, bzw. welcher Hint in welchem Falle der richtige ist. Ich werde beschreiben, wie man Hints unsichtbar in ein SQL Statement einschleusen kann, ohne das SQL und seine SQL ID zu verändern. Dazu beschreibe ich 4 verschiedene Wege, die Nutzung von Stored Outlines, SQL Profiles, SQL Patches, SQL Plan Baselines für dieses Unterfangen. Das Beispiel, welches ich jeweils verwenden werde, ist das folgende:
Query ohne Hints:

```
SELECT count(*) FROM t WHERE id=0815;
```

Plan:

```
SELECT * FROM table(dbms_xplan.display_cursor(format=>'BASIC'))
-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
```

```
|   1 |  SORT AGGREGATE    |      |
|   2 |   INDEX UNIQUE SCAN| T_PK |
-----------------------------------
```

Alternative Query mit Hint:

```
SELECT /*+ FULL(t) */ count(*) FROM t WHERE id=0815
```

Plan der Query mit Hint:

```
SELECT * FROM table(dbms_xplan.display_cursor(format=>'BASIC'))
-----------------------------------
| Id  | Operation          | Name |
-----------------------------------
|   0 | SELECT STATEMENT   |      |
|   1 |  SORT AGGREGATE    |      |
|   2 |   TABLE ACCESS FULL| T    |
-----------------------------------
```

Sie könnten jetzt argumentieren, dass diese Art des Eingriffs in den Ausführungsplan keinen Sinn macht. Aber darum geht es nicht, es geht lediglich um die Illustration des Einflusses auf den Optimizer mit einem einfachen Beispiel.
Die Herausforderung ist es nun also, diesen `FULL` Hint einzubringen, ohne das SQL und die SQL ID (es ist `btuu7yga9bcp1`) zu ändern.

Stored Outlines

Stored Outlines sind das älteste Planstabilitäts-Feature der Oracle Datenbank. Es gibt sie seit Oracle 8i und sie wurden seit Oracle 11g deprecated, sind aber auch noch in Oracle 12c supported und funktionieren in dieser Version natürlich auch noch. Sie können auch in Oracle Express Edition und Standard Edition (One) verwendet werden und benötigen kein Tuning Pack. Stored Outlines speichern eine volle Menge an Hints für ein SQL Statement und schaffen so einen fixen Ausführungsplan. Die Assoziierung zwischen Statement und Outline erfolgt nach einer Normalisierung, bei welcher das Statement intern (bis auf Literale) in Großbuchstaben konvertiert wird, außerdem wird jeder Whitespace entfernt, Kommentare jedoch nicht. Die Outlines und ihre Hints werden in Tabellen im `OUTLN` Schema gespeichert und können über die Views `cdb_/dba_/all_/user_outlines` bzw. `cdb_/dba_/all_/user_outline_hints` abgefragt werden.
Um sie nach ihrer Erstellung zu benutzen, setzt man einmal pro Session oder systemweit den Parameter `use_stored_outlines`. Meistens wird man ihn auf `TRUE` stellen, es ist aber genauso denkbar, für verschiedene Workloads verschiedene Pläne in verschiedenen Outlines verschiedener Kategorien einzuteilen, in dem Falle würde man es auf die jeweils gewünschte Kategorie stellen. Weitere Details verrät die Oracle Dokumentation oder eine Menge Literatur, inklusive der in der Referenz beschriebenen Dokumente. Hier möchte ich nur auf ein merkwürdiges Phänomen hinweisen. Es ist nicht möglich diesen Parameter dauerhaft in init.ora bzw. spfile zu hinterlegen. Es bedarf schon eines `AFTER STARTUP ON DATABASE` Triggers, um ihn auch über Instance Restarts hinweg dauerhaft zu setzen.
Nun zu unserem Beispiel:
Zuerst erzeugen wir 2 Outlines, eine vom ursprünglichen Statement ohne Hints und eine von dem Statement mit dem Hint:

```
CREATE OUTLINE my_outline ON
SELECT count(*) FROM t WHERE id=0815;

CREATE OUTLINE my_outline_interim ON
SELECT /*+ FULL(t) */ count(*) FROM t WHERE id=0815;
```

Jetzt tauschen wir die den Outlines zugeordneten Hints. Die Hints der ersten Outline werden der zweiten zugeordnet, und umgekehrt:

```
UPDATE outln.ol$hints
SET ol_name=decode(ol_name,'MY_OUTLINE','MY_OUTLINE_INTERIM',
                           'MY_OUTLINE_INTERIM','MY_OUTLINE')
WHERE ol_name IN ('MY_OUTLINE','MY_OUTLINE_INTERIM');

UPDATE outln.ol$
SET hintcount = (
  SELECT  count(*)
  FROM    outln.ol$hints
  WHERE   outln.ol$.ol_name = outln.ol$hints.ol_name
)
WHERE outln.ol$.ol_name in ('MY_OUTLINE','MY_OUTLINE_INTERIM');

COMMIT;
```

Auweia! DML auf Tabellen im OUTLN Schema! Darf man das? Oder ist das ein ganz böser Hack? Ich kann Sie beruhigen. Es ist durch den Oracle Support abgesegnet: “My Oracle Support - How to Specify Hidden Hints (Outlines) on SQL Statements in Oracle 8i (Doc ID 92202.1)”.
Eine noch ausführlichere Beschreibung zu dem Thema wurde von Jonathan Lewis angefertigt: http://www.jlcomp.demon.co.uk/04_outlines.rtf. Dass dieses Dokument im Rich Text Format vorliegt, zeigt, wie alt diese Technik schon ist.
Es ist also wirklich durch Oracle genehmigt, so vorzugehen.
Ich möchte noch anmerken, dass die Outline, die wir oben erstellt haben für andere Literalwerte als „0815“ nicht herangezogen wird, nicht einmal für den äquivalenten Wert „815“. So ein literal-insensitives Verhalten wäre nur mit expliziten Bind-Variablen bzw. durch künstlich erzeugte Bind-Variablen durch erzwungenes Cursor Sharing mittels `cursor_sharing=FORCE` zu erreichen.

SQL Profiles

Ein anderes Plan Stabilitäts-Feature sind die SQL Profiles. Sie wurden mit Oracle 10g eingeführt. Für ihre Verwendung ist das Tuning Pack und damit die Oracle Enterprise Edition zwingend erforderlich. Auch SQL Profiles helfen dem Optimizer, in dem sie ein paar (aber nicht alle!) Hints für ein bestimmtes SQL Statement speichern. Normalerweise werden sie durch den SQL Tuning Advisor erstellt, welcher am einfachsten über den Oracle Enterprise Manager bedient werden kann. Es ist aber auch möglich, sie manuell zu erstellen, was ich nachfolgend für unsere Zwecke demonstrieren werde. Ebenso wie bei Stored Outlines erfolgt die Zuordnung von Statement zu Profile nach Normalisierung. Aber im Gegensatz zu Stored Outlines ist bei SQL Profiles auch eine literal-insensitive Zuordnung konfigurierbar. Die Hints für SQL Profiles sind in den Tabellen `sys.sqlobj$`, `sys.sqlobj$auxdata`, `sys.sql$text`, `sys.sql$` gespeichert (das war nicht immer dort, die Aussage in diesem Satz gilt zumindest für Oracle 12c). Als Data Dictionary View für die Übersicht über die im System vorhandenen Profile gibt es `cdb_/dba_sql_profiles`, eine `all_` bzw. `user_` View entfällt, da sie nur für die Verwendung durch DBA-like User gedacht sind.
Und nun unser Beispiel:
Wir erzeugen ein SQL Profile, was unseren `FULL` Hint enthält, außerdem können wir dieses Mal einen literal-insensitiven Match (durch `force_match=>TRUE`) konfigurieren:

```
BEGIN
  dbms_sqltune.import_sql_profile(
    name        => 'MY_SQL_PROFILE',
    description => 'full table scan',
    sql_text    => 'SELECT count(*) FROM t WHERE id=0815',
    force_match => true,
    profile     => sqlprof_attr('FULL(@"SEL$1" "T"@"SEL$1")')
  );
```

```
END;
```
Sie werden sich jetzt sicher fragen: Wie kommt man auf diese seltsame Syntax rund um den `FULL` Hint? Nun – die können Sie sich sogar erzeugen lassen. Lassen Sie dazu einfach das Statement mit Hint ablaufen und den Plan inklusive seiner internen Outline anzeigen. Das geht so:
```
SELECT /*+ FULL(t) */ count(*) FROM t WHERE id=0815;
SELECT * FROM table(dbms_xplan.display_cursor(format=>'ADVANCED'));
```
Achten Sie dann in der Ausgabe auf den Abschnitt „Outline Data“. Dieser sollte ungefähr (je nach Release bzw. Optimizer Umgebung) so aussehen:
```
Outline Data
-------------

  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('12.1.0.2')
      DB_VERSION('12.1.0.2')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      FULL(@"SEL$1" "T"@"SEL$1")
      END_OUTLINE_DATA
  */
```
Die Syntax des voll qualifizierten `FULL` Hints inklusive der Referenz des Query Block Namens finden sie also leicht in dieser Outline Data.
Und nun wieder zur Legitimation dieses Vorgehens: ja, die benutzte Prozedur `import_sql_profile` aus dem an sich dokumentierten PL/SQL Paket `dbms_sqltune` ist selbst nicht dokumentiert. Dennoch gibt es hier einen Ausweg: das Framework rund um das SQL Tuning Werkzeug SQLT, siehe „My Oracle Support - SQLT Diagnostic Tool (Doc ID 215187.1)“, benutzt in den ausgelieferten Skripts – genau gesagt in `sqlt\utl\coe_xfr_sql_profile.sql` – dieselbe Prozedur. In der Beschreibung des Skriptes heißt es, dass es zur Erstellung eines manuellen SQL Profiles für einen bekannten Plan gedacht ist. Natürlich ist der von uns gewünschte Plan bekannt. Das beschriebene Verfahren ist also akzeptiert.

SQL Patches
Das nächste Planstabilitätsfeature, welches wir uns ansehen werden, sind die SQL Patches. Auch damit können wir unser Ziel erreichen. SQL Patches speichern ebenso einzelne Hints, um den Optimizer zu unterstützen. Sie wurden für den Zweck der Fehlervermeidung während der Ausführung von bestimmten Ausführungsschritten von SQL Statements designt. Sie werden normalerweise durch den SQL Repair Advisor erstellt. Hier zeige ich nun auch einen Weg, wie sie manuell erstellt werden können.
SQL Patches und SQL Repair Advisor sind im Gegensatz zu SQL Profiles auch in der Standard Edition (One) und sogar Express Edition vorhanden, dieses beschreibt auch das folgende Dokument von Ulrike Schwinn, Oracle Deutschland: http://www.oracle.com/webfolder/technetwork/de/community/dbadmin/tipps/advisor/index.html.
SQL Patches speichern ebenso nicht alle, sondern nur einen bzw. einige Hints, um die Ausführung eines Statements zu beeinflussen. Die Zuordnung zwischen SQL Statement und SQL Patch erfolgt wie schon vorher gesehen erst nach Normalisierung. Und auch für SQL Patches ist literal-insensitiver Match konfigurierbar. Interessanterweise speichern SQL Patches ihre Hints in exakt denselben Tabellen wir SQL Profiles, nämlich in `sys.sqlobj$, sys.sqlobj$auxdata, sys.sql$text, sys.sql$`. Sie sind dort nur mit einem besonderen Flag als Patch gekennzeichnet. Als Views stehen `cdb_sql_patches` und `dba_sql_patches` bereit. Auch hier gibt es keine `all_` und `user_` Variante.

Und nun zu unserem Beispiel. Wir bringen einen Hint in einen manuell erstellten SQL Patch ein:

```
BEGIN
  sys.dbms_sqldiag_internal.i_create_patch(
      sql_text  => 'SELECT count(*) FROM t WHERE id=0815',
      hint_text => 'FULL(@"SEL$1" "T"@"SEL$1")',
      name      => 'MY_SQL_PATCH');
END;
```

Auf diese Methode machte das Oracle Optimizer Team rund um Maria Colgan, der früheren Product Managerin des Oracle Optimizer, in seinem Blog aufmerksam. Wenn man diese Methode dann in einen provozierten Fehler laufen lässt, so kann man aus der Fehlermeldung schnell ablesen, welches Paket durch `sys.dbms_sqldiag_internal.i_create_patch` gekapselt wurde:

```
...
ERROR at line 1:
ORA-01948: identifier's name length (129) exceeds maximum (30)
ORA-06512: at "SYS.DBMS_SQLTUNE_INTERNAL", line 17518
ORA-06512: at "SYS.DBMS_SQLDIAG_INTERNAL", line 247
ORA-06512: at line 2
```

Wenn man sich dieses Paket nun genauer ansieht fällt einem die Funktion `i_create_sql_profile` mit dem Argumenten `force_match` und `is_patch` auf. Wie sie zu verwenden ist, liegt dann auf der Hand. Es sieht also etwas komplizierter aus, wenn wir einen literal-insensitiven Match für unser Beispiel erzwingen möchten:

```
DECLARE
  v_name VARCHAR2(128);
BEGIN
  v_name :=
    sys.dbms_sqltune_internal.i_create_sql_profile(
      sql_text    => 'SELECT count(*) FROM t WHERE id=0815',
      profile_xml => '<outline_data><hint>
                       <![CDATA[FULL(@"SEL$1" "T"@"SEL$1")]]>
                      </hint></outline_data>',
      is_patch    => true,
      name        => 'MY_SQL_PATCH',
      force_match => true);
END;
```

Ist dieses Vorgehen akzeptiert? Nun ja, wie gesagt, hat das Oracle Optimizer Team in ihrem Blog auf diese Möglichkeit hingewiesen.

SQL Plan Baselines

Das letzte vorgestellte Planstabilitätsfeature sind die SQL Plan Baselines. Sie sind zugleich die von Oracle als die zukunftsträchtigste dargestellte Funktionalität für stabile Pläne. Sie bietet als einzige die Möglichkeit, adäquat auf folgende Datenänderungen reagieren zu können. Bevor das genauer gezeigt werden kann, müssen wir zunächst mal die SQL Plan Baselines erklären.

SQL Plan Baselines sind Teil des SQL Plan Management Features. Sie sind Objekte, die für ein SQL Statement nach Normalisierung, Ausführungsplan und Ausführungsstatistiken speichern. Sobald es eine Baseline für ein Statement, damit also mindestens einen akzeptierten Plan gibt, werden weitere generierte Pläne des Optimizers zunächst nicht akzeptiert. Sie landen zur weiteren Untersuchung, die Oracle Plan Evolution nennt, in der Plan Historie. Wenn während einer Evolution, das ist das Ausprobieren des Planes und das Vergleichen der genutzten Ressourcen und Laufzeiten mit der Baseline, ein bisher nicht akzeptierter Plan ein besseres Ergebnis zeigt, so wird er akzeptiert. Und genau dieses Vorgehen ermöglicht uns, auf spätere Datenumverteilungen oder -änderungen reagieren zu können. Pläne in einer Baseline bzw. Plan Historie können akzeptiert also verifiziert, fixiert oder enabled bzw. disabled sein. Es besteht keine Notwendigkeit für das Tuning Pack, um SQL Plan

Baselines zu nutzen, jedoch funktionieren sie nur in der Enterprise Edition. Es gibt aber einen ganz gravierenden Nachteil zu SQL Profiles und SQL Patches, es ist leider kein literal-insensitiver Match konfigurierbar. Und wo sind nun die Hints der SQL Plan Baselines gespeichert, natürlich wiederum in `sys.sqlobj$`, `sys.sqlobj$auxdata`, `sys.sql$text`, `sys.sql$`, übrigens dieses mal wieder die volle Menge, so wie auch schon bei den SQL Outlines. Zugriff auf Informationen zu SQL Plan Baselines gibt es über die Views `cdb_sql_plan_baselines` und `dba_sql_plan_baselines`.
Nun zurück zu unserem Beispiel: Zunächst müssen wir eine SQL Plan Baseline für das Statement ohne Hint anlegen, welches wir über seine SQL ID referenzieren:

```
DECLARE
  v_cnt NUMBER;
BEGIN
  v_cnt := dbms_spm.load_plans_from_cursor_cache(
             sql_id=>'btuu7yga9bcp1');
END;
```

Jetzt fragen wir SQL Handle, Plan Name von dieser Baseline ab:

```
SELECT sql_handle, sql_text, plan_name, description, enabled
FROM dba_sql_plan_baselines
WHERE sql_text LIKE 'SELECT count(*) FROM t WHERE id=0815'
```

Mit diesen Informationen ausgestattet können wir den Plan erst mal disablen, weil es eben der nicht gewünschte ist. Optional können wir dem Plan noch einen sinnvolleren Namen geben und eine Beschreibung hinzufügen:

```
DECLARE
  v_cnt NUMBER;
BEGIN
  v_cnt := dbms_spm.alter_sql_plan_baseline(
    sql_handle      => 'SQL_d7cd0a6e6c80df97',
    plan_name       => 'SQL_PLAN_dgm8adtq81rwr916941b3',
    attribute_name  => 'enabled',
    attribute_value => 'NO');
  v_cnt := dbms_spm.alter_sql_plan_baseline(
    sql_handle      => 'SQL_d7cd0a6e6c80df97',
    plan_name       => 'SQL_PLAN_dgm8adtq81rwr916941b3',
    attribute_name  => 'plan_name',
    attribute_value => 'NOT_DESIRED_PLAN');
  v_cnt := dbms_spm.alter_sql_plan_baseline(
    sql_handle      => 'SQL_d7cd0a6e6c80df97',
    plan_name       => 'NOT_DESIRED_PLAN',
    attribute_name  => 'description',
    attribute_value => 'original, but not desired plan');
END;
```

Das SQL Handle und den Plan Namen haben wir vorab aus `dba_sql_plan_baselines` abgefragt.
Jetzt müssen wir das Statement mit Hint in den Library Cache laden, indem wir es einmal ausführen:

```
SELECT /*+ FULL(t) */ count(*) FROM t WHERE id=0815
```

Aus `v$sql` ermitteln wir jetzt SQL ID und Plan Hash Value für diese Ausführung:

```
SELECT sql_id, plan_hash_value
FROM v$sql WHERE sql_text LIKE
            'SELECT /*+ FULL(t) */ count(*) FROM t WHERE id=0815'
```

und assoziieren sie mit der vorab schon erstellten SQL Plan Baseline mit dem SQL Handle `SQL_d7cd0a6e6c80df97`:

```
DECLARE
  v_cnt NUMBER;
BEGIN
  v_cnt := dbms_spm.load_plans_from_cursor_cache(
```

```
    sql_id          => 'buzbqk5t2m81y',
    plan_hash_value => '2966233522',
    sql_handle      => 'SQL_d7cd0a6e6c80df97');
END;
```

Der Dokumentation wegen können wir nun diesen zweiten Plan auch noch sinnvoll benennen und eine Beschreibung hinzufügen:

```
DECLARE
  v_cnt NUMBER;
BEGIN
  v_cnt := dbms_spm.alter_sql_plan_baseline(
    sql_handle => 'SQL_d7cd0a6e6c80df97',
    plan_name => 'SQL_PLAN_dgm8adtq81rwr3fdbb376',
    attribute_name => 'plan_name',
    attribute_value => 'DESIRED_PLAN');
  v_cnt := dbms_spm.alter_sql_plan_baseline(
    sql_handle => 'SQL_d7cd0a6e6c80df97',
    plan_name => 'DESIRED_PLAN',
    attribute_name => 'description',
    attribute_value => 'desired plan introduced by the hint');
END;
```

Durch diesen – zugegebener Weise – etwas längeren Weg haben wir also tatsächlich einen vierten Weg gezeigt, wie wir den FULL Hint unsichtbar an unser SQL Statement anheften konnten.
Auch diese Schritte sind durch Maria Colgan in verschiedenen Posts auf http://blogs.oracle.com/optimizer und in ihrer Präsentation „Harnessing the Power of Optimizer Hints“ beschrieben. Durch diese Präsentation kam das Mantra „If you can hint it, baseline it“ auf, was nochmals unterstreicht, dass Baselines am besten geeignet sind, Hints in Queries einzubringen, da durch die Plan Evolution später immer noch eine Änderung zum noch Besseren möglich ist. Alle in diesem Absatz verwendeten Prozeduren sind dokumentiert, daher also absolut genehmigt.

Allgemeines, Vergleich und Fazit

Wir haben nun also 4 Wege gesehen, unseren `FULL` Hint einzubringen. Seine Verwendung sollten wir nun auch noch verifizieren. Das lässt sich am einfachsten folgendermaßen bewerkstelligen:

```
SELECT count(*) FROM t WHERE id=0815;
SELECT * FROM table(dbms_xplan.display_cursor(format=>'basic +note'));
```

In der Notiz des Planes sollte jetzt ein Hinweis auf das verwendete Planstabilitätsfeature stehen. Das wäre dann eine der folgenden, je nachdem für welches Feature Sie sich entschieden haben:

```
   - outline "MY_OUTLINE" used for this statement
   - SQL profile MY_SQL_PROFILE used for this statement
   - SQL patch "MY_SQL_PATCH" used for this statement
   - SQL plan baseline DESIRED_PLAN used for this statement
```

Alle diese Features lassen sich auch in umgekehrten Situationen verwenden, wo eine Applikation mit Statements mit Hints ausgeliefert wird, Sie aber das Ignorieren dieses Hints wünschen. Verwenden Sie in diesem Falle exakt dieselben Schritte, indem Sie einen konträren Hint oder den `IGNORE_OPTIM_EMBEDDED_HINTS` Hint einbringen. Die einfachste Lösung wäre jedoch den Parameter `_optimizer_ignore_hints` auf Session- oder Systemebene auf `TRUE` zu setzen. Für Stored Outlines gibt es hier noch einen Spezialfall, man kann die Liste der gespeicherten Hints auch auf einen reduzieren, nämlich `IGNORE_OPTIM_EMBEDDED_HINTS`, so erreicht man dasselbe Ziel.
Outlines können mittels dbms_spm Package in SQL Plan Baselines migriert werden. Das geht zum Beispiel so:

```
variable v clob
exec :v := dbms_spm.migrate_stored_outline('outline_name','MY_OUTLINE');
DROP OUTLINE my_outline;
```

Stored Outlines können mittels Data Pump von DB zu DB übertragen werden, hier verweise ich gerne auf das Buch Troubleshooting Oracle Performance von Christian Antognini für eine ausführliche Anleitung. SQL Profiles, SQL Patches und SQL Plan Baselines können auch übertragen werden. Dazu müssen sie vorab allerdings in eine Stage-Tabelle gepackt werden. Dazu verwendet man die Pack-Prozeduren aus `dbms_sqltune`, `dbms_sqldiag` bzw. `dbms_spm`. Zum Erstellen nutzt man zunächst:

```
dbms_sqltune.create_stgtab_sqlprof
dbms_sqldiag.create_stgtab_sqlpatch
dbms_spm.create_stgtab_baseline
```

Zum Ein- und Auspacken dann:

```
dbms_sqltune.(un)pack_stgtab_sqlprof
dbms_sqldiag.(un)pack_stgtab_sqlpatch
dbms_spm.(un)pack_stgtab_baseline
```

Das Interessante hier ist, dass die Tabellen für alle drei Fälle dieselbe Struktur aufweisen. Sie können daher sogar ein und dieselbe Stage-Tabelle verwenden, um SQL Profiles, SQL Patches und SQL Plan Baselines hineinzupacken.

Wie in der Einleitung erwähnt, gibt es einige Hints, für die es keine wirkliche Alternative und auch keinen klügeren Optimizer gibt, z.B. `APPEND`, `GATHER_PLAN_STATISTICS`; `RESULT_CACHE`. Nun, es gelingt tatsächlich, auch diese Hints unsichtbar einzubringen. `GATHER_PLAN_STATISTICS` lässt sich durch SQL Profiles oder durch SQL Patches einschleusen und kann sehr hilfreich bei Tuning Sessions sein, um mehr über die Ausführung eines Statements zu erfahren. `RESULT_CACHE` funktionierte in meinen Tests für SQL Patches, aber auch nur für diese. `APPEND` funktionierte mit SQL Profiles und SQL Patches.

Um die dargestellten Techniken anzuwenden, sollte man administrative Rechte haben. Genaue Details gibt die Dokumentation. Ich möchte hier mit dieser groben Aufstellung helfen:

Stored Outlines:

```
create any outline, alter any outline, drop any outline
```

DML Privilegien auf 3 `OUTLN` tables

SQL Profiles, SQL Patches, SQL Plan Baselines:

```
administer sql management object
```

Für einige gezeigte Fälle: `execute on sys.dbms_diag_internal, sys.dbms_sqltune_internal`

Es ist sinnvoll das `select any dictionary` Recht zu verlangen, um alle relevanten Data Dictionary Views abfragen zu können, alternativ reichen natürlich auch Select-Rechte auf alle in diesem Dokument erwähnten Views.

Hier nun ein abschließender Vergleich der verwendeten Technologien:

	Stored Outlines	SQL Profiles	SQL Patches	SQL Plan Baselines
Verfügbar seit	8i	10g	11g	11g
Deprecated?	Seit 11g	Nein	Nein	Nein
XE/SE/SE1	Ja	Nein	Ja	Nein
Tuning Pack benötigt	Nein	Ja	Nein	Nein
Match nach Normalisierung	Ja	Ja	Ja	Ja
Literal-insensitiver Match konfigurierbar	Nein	Ja	Ja	Nein
Datenveränderungen werden respektiert	Nein	Etwas	Etwas	Ja
Kann verwendet werden, um Hints außer Kraft zu setzen	Ja	Ja	Ja	Ja
Verschiedene Kategorien für verschiedene Workloads	Ja	Ja	Ja	Nein

konfigurierbar				
`GATHER_PLAN_STATISTICS` Hint funktioniert	Nein	Ja	Ja	Nein
`RESULT_CACHE` Hint funktioniert	Nein	Nein	Ja	Nein
Kann verwendet werden, um in Parallel Execution einzugreifen	Ja	Ja	Ja	Ja

In diesem abschließenden Vergleich sieht man deutlich, dass es keinen einzigen Favoriten gibt. Je nach Situation gibt es ein am besten passendes Feature. Erstaunlicherweise schneiden SQL Patches im Gesamtvergleich sehr positiv ab.
Bevor Sie eine der gezeigten Techniken in Ihrer Produktion anwenden, sollten Sie es sorgfältig in Ihrem Testsystem getestet haben. Und denken Sie daran, dass Hints in den meisten Fällen nur Workarounds sind.

Viele stabile Pläne wünscht Mathias Zarick.

Quellennachweis

[1] Maria Colgan - How do I migrate stored outlines to SQL Plan Management? - https://blogs.oracle.com/optimizer/entry/how_do_i_migrate_stored

[2] Maria Colgan - How do I deal with a third party application that has embedded hints that result in a sub-optimal execution plan in my environment? - https://blogs.oracle.com/optimizer/entry/how_do_i_deal_with_a_third_party_application_that_has_embedded_hints_that_result_in_a_sub-optimal_ex

[3] Maria Colgan - Oracle Database Optimizer: Harnessing the Power of Optimizer Hints - http://www.nocoug.org/download/2012-11/NoCOUG_201211_Maria_Colgan_Optimizer_Hints.pdf

[4] Allison / Oracle Optimizer Blog - Additional Information on SQL Patches - https://blogs.oracle.com/optimizer/entry/additional_information_on_sql_patches

[5] Allison / Oracle Optimizer Blog - What should I do with old hints in my workload? - https://blogs.oracle.com/optimizer/entry/what_should_i_do_with_old_hints_in_my_workload

[6] Allison / Oracle Optimizer Blog - Using SQL Patch to add hints to a packaged application - https://blogs.oracle.com/optimizer/entry/how_can_i_hint_a

[7] Christian Antognini - SQL Profiles - http://antognini.ch/papers/SQLProfiles_20060622.pdf

[8] Christian Antognini - Troubleshooting Oracle Performance, 2nd Edition - Apress, ISBN-13: 978-1-4302-5758-5

[9] My Oracle Support - How to Specify Hidden Hints (Outlines) on SQL Statements in Oracle 8i (Doc ID 92202.1)

[10] Enkitec Blog - http://blog.enkitec.com/enkitec_scripts/exchange_outline_hints.sql

[11] Jonathan Lewis - Plan Stability in Oracle 8i/9i - http://www.jlcomp.demon.co.uk/04_outlines.rtf

[12] Jonathan Lewis - Hints on Hints - http://jonathanlewis.files.wordpress.com/2009/05/hints_on_hints.pdf

[13] Jonathan Lewis - Rules for Hinting - http://jonathanlewis.wordpress.com/2008/05/02/rules-for-hinting/

[14] Kerry Osborne - Licensing Requirements for SQL Profiles - http://kerryosborne.oracle-guy.com/2011/01/licensing-requirements-for-sql-profiles/

Kontaktadresse:
Mathias Zarick
Trivadis Delphi GmbH
Millennium Tower
Handelskai 94-96
A-1200 Wien

Telefon:	+43 (0) 664 85 44 295
Fax:	+41 (0) 1 332 35 34
E-Mail	Mathias.Zarick@trivadis.com
Internet:	www.trivadis.com