Oracle Old Features

Uwe Küchler
Valentia GmbH
Frankfurt am Main

Schlüsselwörter:

Datenbank, Performance, Constraints, ANSI SQL, PL/SQL.

Einleitung

Bereits im vorigen Jahrtausend hat Oracle Features in ihr RDBMS eingebaut, die heute so nützlich sind wie damals und die dennoch unbekannt oder unterbewertet sind.
In diesem Artikel soll es daher einmal nicht um coole, neue Features gehen sondern für die zu Unrecht vernachlässigten, alten Features eine Lanze gebrochen werden. Diese betreffen vor allem Entwickler, Datenmodellierer und DBAs im Entwicklungs-Umfeld.
Wir betrachten dabei u.a.

- zwei Geheimwaffen im Grabenkampf zwischen Befürwortern und Gegnern von Integrity Constraints; Achtung: Der Autor ist parteiisch und zeigt auf, warum Constraints auch bei großen Datenmengen für die Performance förderlich statt hinderlich sind.
- ein ANSI-SQL-Feature, das nicht nur Kompatibilität sondern vor allem Codeersparnis und Performancegewinne bringt;
- eine ANSI-SQL-Funktion, die den Umgang mit NULLs komfortabler und dabei auch noch schneller gestaltet und
- warum die böse, alte Cursor-FOR-LOOP doch gar nicht so lahm ist.

Constraints – Beschränkung und Beschleunigung zugleich

„*Wir setzen keine Constraints in unserer [großen Datenbank|DWH-DB|...] ein, weil die Performance dann zu schlecht ist*“ – diese Ausrede gilt spätestens seit Oracle 8i nicht mehr. Oft werden dem unbestrittenen Nutzen von Constraints Nachteile im Zeitbedarf gegenüber gestellt, dabei lassen sich diese Nachteile oftmals leicht umschiffen. Zudem kann die Geschwindigkeit von Abfragen wesentlich schlechter werden, wenn *keine* Constraints eingesetzt werden!

Rückblick

Constraints schützen nicht nur die *Integrität* Ihrer Daten, sie enthalten als Teil des Datenmodells auch Informationen über den *Aufbau* der Daten. Diese Informationen sind nicht nur für Anwendungsentwickler nützlich sondern dienen dem Optimizer auch dazu, den besten Ausführungsplan für eine Abfrage zu finden. Rekapitulieren wir kurz die möglichen Arten von Constraints und ihren Informationsgehalt:

- **Primärschlüssel** gewährleisten, daß die Inhalte der zugehörigen Spalten einen Datensatz eindeutig identifizieren. Anders als bei einem reinen Unique Key Constraint sind NULL-Werte ausgeschlossen.
- **Eindeutigkeitsschlüssel** oder auch Unique Key Constraints gewährleisten die Eindeutigkeit über die gewählten Spalten. NULL-Werte sind hier zulässig.
- **NOT NULL und Check-Constraints** beschränken die Inhalte einer Spalte. Während NOT NULL die klare Aussage über ein Pflichtfeld enthält, gehen Check-Constraints noch weit darüber hinaus: Wird bei der Abfrage einer Spalte mit Check-Constraints ein nicht im Constraint enthaltener Wert benutzt, dann kann der Optimizer sofort, d.h. ohne Tabellen- oder Index-Zugriffe, eine leere Menge zurück liefern (ein leider besonders häufig übersehenes Feature).
- **Fremdschlüssel** sorgen nicht nur für die Konsistenz von Daten über abhängige Tabellen hinweg, sie liefern dem Optimizer wiederum Informationen, die er einfach nicht erraten kann: Den Zusammenhang mehrerer Tabellen untereinander und insbesondere die Sicherheit, daß zu einem (NOT NULL-) Wert in einer Kind-Tabelle definitiv ein Wert in der Eltern-Tabelle existiert.

Beispiele

Mit Hilfe der Autotrace-Funktion sollen die folgenden Beispiele demonstrieren, wie sich Constraints auf die SQL-Ausführung auswirken. Zwecks Übersichtlichkeit sind im SQL-Text einige Antworten des DB-Servers ausgelassen. Ferner wurden die Abfragen mindestens zweimal durchgeführt, damit in den Statistiken Sondereffekte durch rekursives SQL bereinigt werden.
Beginnen wir mit NOT NULL und CHECK-Constraints:

```
CREATE TABLE demo
(
  id NUMBER NOT NULL
, name VARCHAR2(50)
, typ VARCHAR2(1)
, CONSTRAINT demo_typ_cc CHECK ( typ IN ('a','b') )
);

INSERT INTO demo VALUES( 1, 'Asterix', 'a' );
INSERT INTO demo VALUES( 2, 'Oraculix', 'b' );

SET AUTOTRACE TRACEONLY

SELECT count(*) FROM demo WHERE name IS NULL;

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          7  consistent gets
          0  physical reads
```

Die Statistik zeigt, daß bei der Abfrage mit NOT NULL auf eine Spalte ohne Constraint Logischer I/O (LIO) auf die Tabelle erfolgt. Fragen wir nun die Spalten mit Constraints ab:

```
SELECT count(*) FROM demo WHERE id IS NULL;
```

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     1 |    13 |     0   (0)|          |
|   1 |  SORT AGGREGATE     |      |     1 |    13 |            |          |
|*  2 |   FILTER            |      |       |       |            |          |
|   3 |    TABLE ACCESS FULL| DEMO |     2 |    26 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(NULL IS NOT NULL)


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          0  consistent gets
          0  physical reads

SELECT count(*) FROM demo WHERE typ='c';

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          0  consistent gets
          0  physical reads
```

Hier zeigt sich nun in beiden Fällen der Vorteil der Constraints: „NOT NULL“ sowie ein nicht vom Check-Constraint erlaubter Wert im Filter sorgen dafür, daß die Abfrage ohne LIO durchgeführt wird. Auffällig ist nebenbei die im Ausführungsplan aufgeführte Filteroperation „NULL IS NOT NULL“, an der man ablesen kann, daß der Optimizer die Möglichkeit zur Vereinfachung der Abfrage wahrgenommen hat.

Kommen wir nun zu den Fremdschlüsseln. Dazu bauen wir zu der Tabelle „demo“ noch eine Tabelle mit Details zur Spalte „typ“, aber zunächst ohne Fremdschlüssel:

```
CREATE TABLE demo_typ
(
  typ VARCHAR2(1) NOT NULL
, detail VARCHAR2(50) NOT NULL
, CONSTRAINT typ_pk PRIMARY KEY ( typ )
);

INSERT INTO demo_typ VALUES( 'a', 'Gallier' );
INSERT INTO demo_typ VALUES( 'b', 'Informatiker' );
```

Wenn wir nun die Tabellen „demo“ und „demo_typ“ mit einem Join abfragen, so wird dieser Join auch dann ausgeführt, wenn gar keine Informationen aus „demo_typ“ verlangt waren:

```
SELECT name FROM demo NATURAL JOIN demo_typ;
```

```
-----------------------------------------------------
|   0 | SELECT STATEMENT   |         |     2 |    62 |
|   1 |  NESTED LOOPS      |         |     2 |    62 |
|   2 |   TABLE ACCESS FULL| DEMO    |     2 |    58 |
|*  3 |   INDEX UNIQUE SCAN| TYP_PK  |     1 |     2 |
-----------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("DEMO"."TYP"="DEMO_TYP"."TYP")
       filter("DEMO_TYP"."TYP"='a' OR "DEMO_TYP"."TYP"='b')
```

Nun fügen wir einen Fremdschlüssel von „demo“ zu „demo_typ“ hinzu und führen die Abfrage erneut aus:

```
ALTER TABLE demo ADD
 CONSTRAINT dem_demtyp_fk FOREIGN KEY ( typ ) REFERENCES demo_typ ( typ );

SELECT name FROM demo NATURAL JOIN demo_typ;

-------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes |
-------------------------------------------------
|   0 | SELECT STATEMENT  |      |     2 |    58 |
|*  1 |  TABLE ACCESS FULL| DEMO |     2 |    58 |
-------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("DEMO"."TYP" IS NOT NULL)
```

Der Optimizer hat also erkannt, daß der Join zu „demo_typ“ gar nicht benötigt wird und führt ihn daher auch nicht durch („Table Elimination“). Dieses Feature kommt besonders dann zum Tragen, wenn Abfragen nicht direkt über Tabellen, sondern über Views gemacht werden, in denen Joins und abgefragte Spalten schon vorgegeben sind.

Prüfen wir abschließend, was passiert, wenn Check-Constraints zu Table Elimination führen müssten:

```
SELECT name, detail
  FROM demo d, demo_typ t
 WHERE d.typ = t.typ
   AND d.typ = 'c';

------------------------------------------------------------------------
| Id  | Operation                      | Name     | Rows  | Bytes |
------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |          |     1 |    58 |
|*  1 |  FILTER                        |          |       |       |
|   2 |   MERGE JOIN CARTESIAN         |          |     1 |    58 |
|*  3 |    TABLE ACCESS FULL           | DEMO     |     1 |    29 |
|   4 |    BUFFER SORT                 |          |     2 |    58 |
```

```
|   5 |      TABLE ACCESS BY INDEX ROWID| DEMO_TYP |      2 |     58 |
|*  6 |       INDEX UNIQUE SCAN         | TYP_PK   |      1 |        |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(NULL IS NOT NULL AND NULL IS NOT NULL)
   3 - filter("D"."TYP"='c')
   6 - access("D"."TYP"="T"."TYP")

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          0  consistent gets
          0  physical reads
          0  redo size
        344  bytes sent via SQL*Net to client
        408  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          0  rows processed
```

Der von Autotrace generierte Ausführungsplan lässt zwar zunächst anderes vermuten, aber die Filteroperation in Schritt 1 sowie die Statistiken beweisen es: Der Optimizer kombiniert beide Möglichkeiten und liefert die leere Menge zurück, ohne LIO zu verursachen und ohne für Joins CPU zu verbrauchen.
Übertragen Sie dieses Wissen von der Laborsituation auf sehr große Fakten- mit vielen, großen Dimensionstabellen, die womöglich mit leistungshungrigen Hash- oder Merge-Joins durchgeführt werden, dann können Sie sich das Potential von Constraints ausmalen!

> „*Das ist ja alles schön und gut, aber die Zeit, um diese Constraints alle aufzubauen und aufrecht zu erhalten, haben wir einfach nicht*“.

Nun – selbst wenn der oft enorme Zeitgewinn durch schneller und schlanker laufende Abfragen allein noch nicht ausreichen sollte, gibt es auch für diese Problematik mehrere Ansätze, bei deren Einsatz man allerdings auch unbedingt wissen sollte, welche Konsequenzen sich daraus ergeben. Dies ist das Thema der folgenden Unterkapitel.

NOVALIDATE

Diese Option soll hier nur kurz im Interesse der Vollständigkeit und zum Vergleich mit den weiter unten aufgeführten Möglichkeiten aufgezeigt werden.
Wenn Sie sich sicher sind, daß die Beladung einer Tabelle konsistent abgelaufen ist, können Sie Constraints mit der Option „ENABLE NOVALIDATE” auf dieser Tabelle aktivieren, ohne daß die bestehenden Inhalte dabei verifiziert werden. Alle späteren Änderungen werden jedoch verifiziert.
Diese Option kommt allerdings mit Einschränkungen:

- Für die Unterstützung von Constraints werden ausschließlich Non-Unique-Indizes angelegt (Unique würde ja gerade die Überprüfung der Inhalte voraussetzen).

- Da Oracle nun nicht mehr absolut sicher „weiß", ob die Inhalte tatsächlich konsistent sind, greifen auch die o.g. Vorteile von NOT NULL- und Check-Constraints nicht.

Diese Einschränkungen betreffen auch Deferrable Constraints und werden weiter unten genauer dargestellt.

RELY-Constraints

Wenn Sie Ihrer/Ihren Applikation/en so sehr vertrauen, daß diese völlig konsistente Daten liefern oder Sie sich bei Bestandsdaten auf deren Konsistenz verlassen können, dann können Sie dieses schon seit Oracle 8i verfügbare Attribut nutzen.
Hier eine Aufstellung der Eigenschaften und Einsatzmöglichkeiten von RELY-Constraints:

- Ein RELY-Constraint wird nicht „enforced", Oracle verlässt sich („rely") auf die Angabe des Designers, dass die Daten valide sind. Das Constraint steht also nur informativ im Data Dictionary, ohne bei INSERTs oder UPDATEs aktiv zu werden.
- RELY kann auf fast alle Arten von Constraints angewandt werden, außer NOT NULL.
- Ein RELY auf Fremdschlüssel setzt voraus, daß der referenzierte Primärschlüssel ebenfalls im RELY-Modus ist.
- Hilft dem Optimizer (und auch Entwicklern), die Beziehungen zwischen Tabellen zu verstehen.
    - Wesentliche Voraussetzung für Query Rewrite, also ein wichtiges Einsatzszenario bei Materialized Views
    - Voraussetzung für Table Elimination (Beispiele s.o.)
- Praktisch im DWH-Umfeld, wenn Integrität bereits durch Beladeprozesse gewährleistet ist und Constraints zu viel Aufwand beim Schreiben bedeuten.
- Können auch auf Views angewandt werden (und zwar *nur* RELY-Constraints!). Wenn Abfragen hauptsächlich über Views laufen, ist dies für den Optimizer nützlich (Query Rewrite).
- Externe Tools können diese Constraints benutzen, um die Zusammenhänge zwischen den Tabellen zu „verstehen". In einem Praxisfall wurden RELY-Constraints eingesetzt, um zunächst das existierende Datenmodell im Schema klarer zu dokumentieren und zum Teil auch schon die Abfrage-Performance zu verbessern (ohne die Performance beim Schreiben zu beeinträchtigen). Anschließend konnte dieses Datenmodell dann per Reverse Engineering in ein Design-Tool eingelesen werden.

Um die Wirkung von RELY-Constraints zu demonstrieren, legen wir unsere Tabellen von oben erneut an:

```
DROP TABLE demo_typ PURGE;
DROP TABLE demo PURGE;
CREATE TABLE demo
(
  id NUMBER NOT NULL ENABLE NOVALIDATE
, name VARCHAR2(50)
, typ VARCHAR2(1)
, CONSTRAINT demo_typ_cc CHECK ( typ IN ('a','b') ) RELY ENABLE NOVALIDATE
);

INSERT INTO demo VALUES( 1, 'Asterix', 'a' );
INSERT INTO demo VALUES( 2, 'Oraculix', 'b' );
```

```
CREATE TABLE demo_typ
(
  typ VARCHAR2(1) NOT NULL
, detail VARCHAR2(50) NOT NULL
, CONSTRAINT typ_pk PRIMARY KEY ( typ ) RELY
);

INSERT INTO demo_typ VALUES( 'a', 'Gallier' );
INSERT INTO demo_typ VALUES( 'b', 'Informatiker' );

ALTER TABLE demo ADD
 CONSTRAINT dem_demtyp_fk FOREIGN KEY ( typ ) REFERENCES demo_typ ( typ )
RELY ENABLE NOVALIDATE;
```

Die Constraints sind innerhalb von Millisekunden angelegt, auch wenn die Tabellen Milliarden von Rows enthalten würden. Die weiter oben demonstrierten Optimizer-Features wie Table Elimination oder Filterung über Check-Constraints können sofort genutzt werden. Noch einmal sei hier erwähnt, daß die Datenkonsistenz nun nicht mehr geprüft wird – im Falle inkonsistenter Daten können dann falsche Ergebnisse geliefert werden!

Ein weiteres, zentrales Optimizer-Feature ist „Query Rewrite“. Es ermöglicht unter anderem, ein SQL im Hintergrund so umzuschreiben, daß anstelle von Tabellen eine Materialized View verwendet werden kann, wenn sie zum ursprünglichen SQL passt:

```
CREATE MATERIALIZED VIEW demo_mv
BUILD IMMEDIATE
REFRESH ON DEMAND
ENABLE QUERY REWRITE
AS
  SELECT t.detail, count (*)
    FROM demo d, demo_typ t
   WHERE d.typ = t.typ
   GROUP BY t.detail;
```

Hier zunächst die einfachste Variante des Query Rewrite: Es wird dieselbe Abfrage wie in der Materialized View verwendet:

```
  SELECT t.detail, count (*)
    FROM demo d, demo_typ t
   WHERE d.typ = t.typ
   GROUP BY t.detail;

DETAIL                                             COUNT(*)
-------------------------------------------------- ----------
Gallier                                                     1
Informatiker                                                1


-----------------------------------------------------------
| Id  | Operation                     | Name    | Rows  |
-----------------------------------------------------------
|   0 | SELECT STATEMENT              |         |     2 |
|   1 |  MAT_VIEW REWRITE ACCESS FULL| DEMO_MV |     2 |
-----------------------------------------------------------
```

Query Rewrite kann aber noch mehr:

```
ALTER SESSION set query_rewrite_integrity='TRUSTED';

SELECT count(*) FROM demo;

-------------------------------------------------------------
| Id  | Operation                     | Name    | Rows  |
-------------------------------------------------------------
|   0 | SELECT STATEMENT              |         |     1 |
|   1 |  SORT AGGREGATE               |         |     1 |
|   2 |   MAT_VIEW REWRITE ACCESS FULL| DEMO_MV |     2 |
-------------------------------------------------------------

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
          0  redo size
        422  bytes sent via SQL*Net to client
        419  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Der Optimizer hat auch diese Abfrage umgeschrieben, da er über folgende Sachverhalte informiert ist:

1. „typ“ ist der Primärschlüssel von „demo_typ“, d.h., jeder Datensatz in „demo“ passt zu maximal einem Datensatz in „demo_typ“.
2. „typ“ in der Tabelle „demo“ hat eine Fremdschlüsselbeziehung zu „demo_typ“.
3. „typ“ in der Tabelle „demo“ ist NOT NULL. Gemeinsam mit dem Fremdschlüssel bedeutet das: Es gibt nicht nur *höchstens* sondern auch *mindestens* eine Entsprechung zu jedem Datensatz von „demo“ in „demo_typ“. Im Umkehrschluss heißt das, daß der COUNT in der Materialized View verwendet werden kann, da durch den Join keine Datensätze aus „demo“ ausgelassen werden.

Überträgt man dieses Beispiel auf große Tabellen in einem DWH, dann sollte nun endgültig belegt sein, welches riesige Potential bei vergleichsweise geringen Kosten in Constraints steckt.

Wie sieht es aber aus, wenn wir im OLTP-Umfeld unterwegs sind oder die Constraints validiert haben möchten? Auch hier gibt es Möglichkeiten, den Overhead von Constraints zu reduzieren, wie das nächste Kapitel zeigt.

Constraint-Prüfung im Batch-Modus (Deferrable Constraints)

Deferrable Constraints sind schon seit Oracle 8 verfügbar. Sie erleichtern den Umgang mit Constraints durch folgende Eigenschaften:

- Ein Deferred Constraint wird nicht sofort beim Insert oder Update geprüft sondern erst beim COMMIT. Dies bringt zum einen bei Massenänderungen Geschwindigkeitsvorteile gegenüber der sonst üblichen Prüfung Zeile für Zeile.
- Zum anderen muss bei einer Beladung keine Rücksicht auf die Reihenfolge von Eltern-Kind-Tabellen genommen werden, wenn die Fremdschlüssel auf "deferred" gestellt sind.
- Schlägt die Prüfung des Constraints fehl, findet ein implizites Rollback statt. Für Transaktionen mit deferred Constraints gilt also ein „alles oder nichts"-Prinzip.
- Zweistufiges Prinzip: Das Constraint muss mit der Option „deferrable" angelegt sein. Mit der Option „initially [immediate | deferred]" wird dann festgelegt, wie das Constraint normalerweise behandelt wird.
- Die Behandlung von deferrable Constraints kann auch auf Session-Ebene umgestellt werden, z.B.: `set constraint all deferred`.

Auch hier sagt ein Beispiel mehr als 1000 Worte:

```
ALTER TABLE demo DROP CONSTRAINT demo_typ_cc;
ALTER TABLE demo DROP CONSTRAINT dem_demtyp_fk;
ALTER TABLE demo ADD
 CONSTRAINT dem_demtyp_fk FOREIGN KEY ( typ ) REFERENCES demo_typ ( typ )
 DEFERRABLE INITIALLY DEFERRED;
```

Das folgende UPDATE würde bei einem herkömmlichen Fremdschlüssel fehlschlagen. Da es aber erst nach dem COMMIT geprüft wird, können die Tabellen in beliebiger Reihenfolge geändert werden:

```
UPDATE demo SET typ = 'g' WHERE typ = 'a';

1 row updated.

UPDATE demo_typ SET typ = 'g' WHERE typ = 'a';

1 row updated.

COMMIT;

Commit complete.
```

Caveats

Primary und Unique Key Constraints werden normalerweise durch Unique Indizes unterstützt. Nicht so im Fall von Deferrable Constraints: Da bei Änderungen an der Tabelle auch die Indizes unmittelbar mit geändert werden, müssen diese Indizes zwangsweise Non-Unique sein, damit vorübergehende Uneindeutigkeiten bis zum Ende der Transaktion überhaupt geduldet werden können.
Dies hat mehrere Implikationen:

1. Non-Unique-Indizes werden zwangsläufig mit Range Scans durchsucht, da Unique Scans verständlicherweise nicht möglich sind. Ergo wird beim Durchsuchen mehr I/O generiert. Darüberhinaus werden mehr Latches benötigt. Beides wirkt sich auf die Performance aus. Dem gegenüber steht eine bessere Wartbarkeit von Non-Unique-Indizes, die hier aber nicht näher betrachtet werden soll.
2. Der Optimizer kann sich nicht mehr darauf verlassen, daß ein Unique Constraint oder ein Primärschlüssel zu jedem Zeitpunkt konsistent ist. Ausführungspläne werden daher in der

Annahme erzeugt, daß inkonsistente Daten vorliegen könnten. Damit sind o.g. Vorteile wie Table Elimination und Filterungen nicht mehr nutzbar.

Die oben erfolgreich demonstrierte Table Elimination geht mit dem Deferrable FK nun nicht mehr:

```
SELECT name FROM demo NATURAL JOIN demo_typ;

-----------------------------------------------
| Id  | Operation          | Name    | Rows  |
-----------------------------------------------
|   0 | SELECT STATEMENT   |         |     2 |
|   1 |  NESTED LOOPS      |         |     2 |
|   2 |   TABLE ACCESS FULL| DEMO    |     2 |
|*  3 |   INDEX UNIQUE SCAN| TYP_PK  |     1 |
-----------------------------------------------
```

Und noch ein Punkt: Wenn Deferrable Constraints vorliegen, können Operationen auf dieser Tabelle nicht parallelisiert werden.

NVLst Du noch oder COALESCEt Du schon?

NULL-Werte an sich können dem DB-Entwickler genug Kopfzerbrechen bereiten, um alleine damit ganze Artikelserien zu füllen. Wir wollen uns hier nur auf einen kleinen Ausschnitt der Problematik beschränken, nämlich der Performance beim Vergleichen und Ersetzen von NULLs. Zum Ersetzen von NULLs in Abfragen wird zumeist die Funktion NVL() verwendet. Diese Funktion hat jedoch zwei entscheidende Nachteile:

1. Sie bietet nur die Möglichkeit, genau einen Ausdruck mit einem anderen zu ersetzen. Will man mehr als zwei Ausdrücke gegeneinander vergleichen, muss die Funktion kaskadiert werden.
2. Der zweite Ausdruck wird *immer* ausgewertet, auch dann, wenn der erste Ausdruck bereits NOT NULL ist.

Hier ein Beispiel , um die möglichen Auswirkungen dieser Nachteile aufzuzeigen: Zur Aufbereitung der Tabelle “employees” sollen alle NULLs durch die Zahl 0 ersetzt werden. Sollte es sich jedoch um einen Verkäufer handeln, soll die standardmäßige Provision von 0,25 % verwendet werden:

```
SELECT employee_id
     , NVL( commission_pct, DECODE( job_id, 'SA_REP', 0.25, 0 )) comm
  FROM employees
```

Wenn wir für dieses SQL ein Trace erzeugen, stellen wir fest, daß das DECODE für jeden Datensatz ausgeführt wird – auch dann, wenn die Spalte “commission_pct” gar nicht NULL ist!

Wenn wir uns nun vorstellen, daß anstelle des trivialen DECODEs ein komplizierteres SELECT oder eine Function steht, die wiederum Abfragen ausführt (z.B. um die Provision in Abhängigkeit des Landes zuzuteilen), wird schnell klar, daß hier sehr viel Arbeit umsonst ausgeführt werden kann.

Kurzschlussreaktion

Seit Oracle-Version 9i steht die Funktion COALESCE zur Verfügung. Gegenüber NVL besitzt sie folgende Vorteile:

1. COALESCE ermöglicht die Auswertung von mehr als zwei Ausdrücken; der erste, der NOT NULL ist, wird zurückgeliefert.
2. COALESCE Nutzt einen sogenannten "Short Circuit", d.h., daß die Funktion beendet wird, sobald der erste Nicht-NULL-Ausdruck gefunden wurde. Bei der Abarbeitung von großen Datenmengen ist dies der entscheidende Vorteil gegenüber NVL.
3. COALESCE ist ANSI-SQL-konform.

Das Beispiel von oben in umgeschriebener Form:

```
SELECT employee_id
     , COALESCE( commission_pct, DECODE( job_id, 'SA_REP', 0.25, 0 )) comm
  FROM employees
```

Es ist nicht verkehrt, komplett von NVL aus COALESCE umzusteigen.

Materialisieren von Unterabfragen: WITH-Klausel

Dieses von Oracle in Version 9i eingeführte ANSI-SQL-Feature hat sich wegen seines Praxisnutzens inzwischen schon etwas herumgesprochen. Die Vorteile gegenüber der alten Schreibweise können aber nicht genug betont werden, also möchte ich dieses „Old Feature" auch hier präsentieren.

Häufig (besonders bei Updates) müssen Spalteninhalte gegen Wertelisten abgegelichen werden. Für diese Listen werden dann Unterabfragen verwendet. Wenn nun in einem SQL dieselbe Werteliste mehrfach benötigt wird, musste bislang die Unterabfrage ebensooft wiederholt werden, und ebensooft musste Oracle dann auf die abgefragten Daten zugreifen.
Mit Einführung der WITH-Klausel kann nun die Zahl der Zugriffe von n auf 1 reduziert und gleichzeitig einige Tipparbeit gespart werden. Früher hätte man in solchen Fällen möglicherweise die Ergebnisse einer aufwendigen Unterabfrage in eine temporäre Tabelle ausgelagert und das äußere SQL dann auf die temporäre Tabelle zugreifen lassen. Nun entscheidet sich der Optimizer zwischen dem internen Umschreiben des SQLs oder der temporären Materialisierung.

Beispielszenario

- Wir wollen aus einer Tabelle „depot_kunde" Daten anzeigen, und zwar in Abhängigkeit von zwei Listen in den Tabellen „tmp_uid" und „tmp_dep".
- Sind Schlüssel aus dem Datensatz in *einer der beiden* Listen enhalten, soll der Datensatz kopiert werden.
- Sind Schlüssel aus dem Datensatz in *beiden* Listen enhalten, soll der Datensatz *nicht* kopiert werden.

Der folgende Code stellt einen möglichen Lösungsansatz vor Oracle 9i dar:

```
CREATE TABLE depot_kunde
(
  kunde_uid NUMBER NOT NULL
, depot_nr NUMBER NOT NULL
, CONSTRAINT depot_kunde_pk PRIMARY KEY ( kunde_uid, depot_nr )
);

CREATE TABLE tmp_uid
(
  kunde_uid NUMBER NOT NULL
```

```
);

CREATE TABLE tmp_dep
(
  depot_nr NUMBER NOT NULL
);

INSERT INTO depot_kunde VALUES( 1, 10 );
INSERT INTO depot_kunde VALUES( 2, 20 );
INSERT INTO depot_kunde VALUES( 3, 30 );
INSERT INTO tmp_dep VALUES( 10 );
INSERT INTO tmp_dep VALUES( 20 );
INSERT INTO tmp_uid VALUES( 2 );
INSERT INTO tmp_uid VALUES( 3 );

SET AUTOT TRACEONLY

SELECT *
  FROM depot_kunde v
 WHERE v.kunde_uid IN( SELECT a.kunde_uid
                         FROM depot_kunde a
                         LEFT OUTER JOIN tmp_uid b ON a.kunde_uid = b.kunde_uid
                         LEFT OUTER JOIN tmp_dep c ON a.depot_nr = c.depot_nr
                        WHERE b.kunde_uid IS NULL
                           OR c.depot_nr IS NULL )
    OR v.depot_nr IN( SELECT a.depot_nr
                        FROM depot_kunde a
                        LEFT OUTER JOIN tmp_uid b ON a.kunde_uid = b.kunde_uid
                        LEFT OUTER JOIN tmp_dep c ON a.depot_nr = c.depot_nr
                       WHERE b.kunde_uid IS NULL
                          OR c.depot_nr IS NULL );

-------------------------------------------------
| Id  | Operation             | Name            |
-------------------------------------------------
|   0 | SELECT STATEMENT      |                 |
|*  1 |  FILTER               |                 |
|   2 |   TABLE ACCESS FULL   | DEPOT_KUNDE     |
|*  3 |   FILTER              |                 |
|*  4 |    HASH JOIN OUTER    |                 |
|*  5 |     HASH JOIN OUTER   |                 |
|*  6 |      INDEX RANGE SCAN | DEPOT_KUNDE_PK  |
|*  7 |      TABLE ACCESS FULL| TMP_UID         |
|   8 |     TABLE ACCESS FULL | TMP_DEP         |
|*  9 |   FILTER              |                 |
|* 10 |    HASH JOIN OUTER    |                 |
|* 11 |     HASH JOIN OUTER   |                 |
|* 12 |      TABLE ACCESS FULL| DEPOT_KUNDE     |
|  13 |      TABLE ACCESS FULL| TMP_UID         |
|* 14 |     TABLE ACCESS FULL | TMP_DEP         |
-------------------------------------------------

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         68  consistent gets
          0  physical reads
```

```
          0  redo size
        529  bytes sent via SQL*Net to client
        419  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed
```

Mit der WITH-Clause kann das doppelt verwendete Subselect nun ausgelagert und vorab materialisiert werden:

```
WITH sub AS
     ( SELECT a.kunde_uid
            , a.depot_nr
         FROM depot_kunde a
         LEFT OUTER JOIN tmp_uid b ON a.kunde_uid = b.kunde_uid
         LEFT OUTER JOIN tmp_dep c ON a.depot_nr = c.depot_nr
        WHERE b.kunde_uid IS NULL
           OR c.depot_nr IS NULL )
SELECT *
  FROM depot_kunde v
 WHERE v.kunde_uid IN( SELECT kunde_uid FROM sub )
    OR v.depot_nr IN( SELECT depot_nr FROM sub );
```

Der Ausführungsplan zeigt nun, daß die Anweisung der With-Clause vorab in eine temporäre Tabelle hinein materialisiert wird. Anschließend greifen die Unterabfragen jeweils auf diese temporäre Tabelle zu:

```
-------------------------------------------------------------------
| Id  | Operation                    | Name                       |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                            |
|   1 |  TEMP TABLE TRANSFORMATION   |                            |
|   2 |   LOAD AS SELECT             | SYS_TEMP_0FD9D6651_27592C  |
|*  3 |    FILTER                    |                            |
|*  4 |     HASH JOIN OUTER          |                            |
|*  5 |      HASH JOIN OUTER         |                            |
|   6 |       INDEX FAST FULL SCAN   | DEPOT_KUNDE_PK             |
|   7 |       TABLE ACCESS FULL      | TMP_UID                    |
|   8 |      TABLE ACCESS FULL       | TMP_DEP                    |
|*  9 |   FILTER                     |                            |
|  10 |    INDEX FAST FULL SCAN      | DEPOT_KUNDE_PK             |
|* 11 |    VIEW                      |                            |
|  12 |     TABLE ACCESS FULL        | SYS_TEMP_0FD9D6651_27592C  |
|* 13 |    VIEW                      |                            |
|  14 |     TABLE ACCESS FULL        | SYS_TEMP_0FD9D6651_27592C  |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("B"."KUNDE_UID" IS NULL OR "C"."DEPOT_NR" IS NULL)
   4 - access("A"."DEPOT_NR"="C"."DEPOT_NR"(+))
   5 - access("A"."KUNDE_UID"="B"."KUNDE_UID"(+))
   9 - filter( EXISTS (SELECT 0 FROM  (SELECT /*+ CACHE_TEMP_TABLE ("T1") */ "C0"
              "KUNDE_UID","C1" "DEPOT_NR" FROM "SYS"."SYS_TEMP_0FD9D6651_27592C" "T1") "SUB"
WHERE
```

```
              "KUNDE_UID"=:B1) OR  EXISTS (SELECT 0 FROM  (SELECT /*+ CACHE_TEMP_TABLE ("T1")
*/ "C0"
               "KUNDE_UID","C1" "DEPOT_NR" FROM "SYS"."SYS_TEMP_0FD9D6651_27592C" "T1") "SUB"
WHERE
             "DEPOT_NR"=:B2))
  11 - filter("KUNDE_UID"=:B1)
  13 - filter("DEPOT_NR"=:B1)
```

Die Informationen über die Prädikate im Ausführungsplan zeigen übrigens noch eine Besonderheit: Das hier gezeigte Beispiel hätte sich auch gut mit der EXISTS- anstelle der IN-Klausel realisieren lassen. Der Optimizer hat dies erkannt und in Schritt 9 das SQL dementsprechend umgeschrieben.
Die Verringerung der Full Scans gegenüber dem ersten Beispielcode zeigt sich auch in den I/O-Statistiken:

```
Statistics
----------------------------------------------------------
          2  recursive calls
          8  db block gets
         37  consistent gets
          1  physical reads
        600  redo size
        529  bytes sent via SQL*Net to client
        419  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          2  rows processed
```

Caveats

- Nicht immer führt ein Auslagern von Unterabfragen in einen WITH-Block automatisch zur Materialisierung diese Blocks. Der einfachste Fall ist, wenn der WITH-Block nur einmal referenziert wird. Es gibt aber auch weitere Szenarien (und Bugs), die eine Materialisierung verhindern. Ein Materialisieren kann mit dem Hint `/*+ MATERIALIZE */` innerhalb des WITH-Blocks erzwungen werden.
- Die Benutzung der WITH-Clause kann in manchen Fällen dazu führen, daß der Optimizer nicht mehr alle Optimierungsansätze durchgeht. Es kann dann gegenüber dem ursprünglichen SQL zu längeren Laufzeiten kommen.
- Es gilt also die alte Regel: Jede Systemkonfiguration hat ihre Eigenheiten, daher sollte jede Lösung ausgiebigen Vergleichstests unterzogen werden.

Kontaktadresse:

Uwe Küchler
Valentia GmbH
Eschersheimer Landstr. 230
60320 Frankfurt am Main

Telefon: +49 (0) 69-1534-1982
Fax: +49 (0) 69-95158934

E-Mail uwe.kuechler@valentia.eu
Internet: www.valentia.eu
Blog: http://oraculix.wordpress.com/