# Oracle Old Features

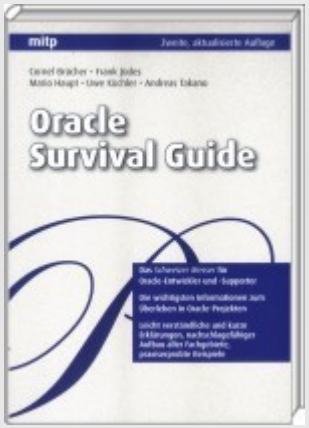
Vortrag für die DOAG-Konferenz 2011

17.11.2011

Uwe M. Küchler, Valentia GmbH

# Zur Person

- Generation C=64
- Seit über 25 Jahren in der IT tätig
- 1997-2000 bei Oracle




- Seither durchgehend Oracle-Berater, im DBA- und Entwicklungs-Umfeld
- Seit 2001: Valentia GmbH, Frankfurt am Main

# Agenda

- Vernachlässigte Features seit Oracle 8.0
- Alt, aber bezahlt: Integrity Constraints
  - Warum wir sie (doch) brauchen
  - Geheimwaffe 1: RELY
  - Geheimwaffe 2: DEFERRABLE
  - Gefahrenherde
- ANSI 1: WITH anstatt Temp-Tabellen
- ANSI 2: COALESCE() vs. NVL()

# Constraints: Contra

Eine unendliche Geschichte:

*„Wir setzen keine Constraints in unserer [großen Datenbank|DWH-DB|...] ein, weil die Performance dann zu schlecht ist“*

Es folgen einige Gegenbeweise!

# Constraints: Rückblick

- Schützen Integrität der Daten, sowohl
  - Inhalte als auch
  - Beziehungen
- Information über den Aufbau der Daten
- Essentiell für den Optimizer
  - Weil er den Aufbau der Daten nicht erraten kann!

# Constraints: Rückblick

- Primärschlüssel
  - Inhalte der zugehörigen Spalten identifizieren einen Datensatz eindeutig
  - NULL ist im Schlüssel ***nicht*** erlaubt
- Eindeutigkeitsschlüssel
  - Eindeutigkeit über die gewählten Spalten
  - NULL ist im Schlüssel erlaubt

# Constraints: Rückblick

- NOT NULL und Check-Constraints
  - beschränken die Inhalte einer Spalte
  - NOT NULL → Pflichtfeld
  - Check-C. begrenzen Inhalte auf Wertelisten →
  - Optimizer „weiß” dadurch, ohne LIO, ob Prädikate leere Mengen zurück liefern.
  - Design-Entscheidung, ob Check-C. die Wartbarkeit beeinträchtigen oder nicht.

# Constraints: Rückblick

- Foreign Key Constraints (Fremdschlüssel)
  - Konsistenz zwischen abhängigen Tabellen
  - Dokumentation der Zusammenhänge im Datenmodell
  - Aussage für den Optimizer, daß zu einem (NOT NULL-) Wert in einer Kind-Tabelle definitiv ein Wert in der Eltern-Tabelle existiert.

# Constraints: Auswirkungen

```
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
```

# Constraints: Auswirkungen

Die vorausgegangene Abfrage auf eine Spalte ohne NOT NULL-Constraint führte zu einem Table Scan, also logischem I/O (LIO).

Fragen wir nun eine Spalte mit Constraint ab:

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

-----------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes |
-----------------------------------------------------
|   0 | SELECT STATEMENT    |      |     1 |    13 |
|   1 |  SORT AGGREGATE     |      |     1 |    13 |
|*  2 |   FILTER            |      |       |       |
|   3 |    TABLE ACCESS FULL| DEMO |     2 |    26 |
-----------------------------------------------------
```

# Constraints: Auswirkungen

```
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
```

Die Abfrage wurde nun ohne LIO durchgeführt.

Die etwas eigenartige Filteroperation “NULL IS NOT NULL” zeigt, daß der Optimizer die Möglichkeit zur Vereinfachung der Abfrage wahrgenommen hat.

# Constraints: Auswirkungen

Nun zur Spalte mit dem Check-Constraint:

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

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

Auch hier erkennt der Optimizer, daß aufgrund des „unmöglichen“ Prädikats kein Tabellenzugriff benötigt wird.

# Constraints: Auswirkungen

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' );
```

# Constraints: Auswirkungen

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')
```

# Constraints: Auswirkungen

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)
```

# Constraints: Auswirkungen

- Table Elimination / Join Elimination
  - Optimizer erkennt, daß ein Join nicht benötigt wird, weil die Spalten einer der Tabellen gar nicht abgefragt werden.
  - Sehr praktisch bei Abfragen über Views.
  - Dazu muß er wissen, daß es in der Eltern-Tabelle mindestens eine und höchstens eine Entsprechung gibt.
  - Genau diese Information beinhaltet ein Fremdschlüssel.

# Constraints: Auswirkungen

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 |       |
-------------------------------------------------------------------
```

Sieht zunächst nicht gut aus, aber:

# Constraints: Auswirkungen

```
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 Optimizer kombiniert beide Möglichkeiten und liefert die leere Menge ohne LIO und CPU-Last für Joins zurück.

# Constraints: Contra, die 2.

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

Es folgen einige Gegenmaßnahmen!

# Constraint-Tuning: NOVALIDATE

- `ALTER CONSTRAINT … ENABLE NOVALIDATE`
- Wenn Konsistenz der Daten bekannt ist
- Prüft die vorhandenen Daten nicht, nur die nachfolgenden.
- Nachteile:
  - Es werden nur Non-Unique Indizes angelegt
  - Vorteile von NOT NULL- und Check-Constraints werden nicht genutzt
  - s.u., bei Deferrable Constraints.

# Constraint-Tuning: RELY

- Seit Oracle 8i
- Ein RELY-Constraint wird nicht „enforced“
  - Oracle vertraut der Angabe des Designers
  - Nur informativ im Data Dictionary
- Alle Arten von Constraints, außer NOT NULL
- RELY auf Fremdschlüssel erfordert RELY auf Primärschlüssel im Ziel.
- Praktisch im DWH-Umfeld, wenn Integrität bereits durch Beladeprozesse gewährleistet ist.

# Constraint-Tuning: RELY

- Können auch auf Views angewandt werden
  - Und zwar *nur* RELY-Constraints!
  - Ermöglicht Query Rewrite durch den Optimizer.
- Kann beim Abfragen nutzen, schadet nicht beim Schreiben
- Anwendung auch zur Dokumentation
  - z.B. per Reverse Engineering in ein Modellierungs-Tool

# Constraint-Tuning: Beispiele

```
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;
```

# Constraints: Query Rewrite

Rewrite: SQL wird im Hintergrund umgeschrieben, so daß z.B. anstelle der Tabellen im SQL auf eine Materialized View zugegriffen wird.
Dazu bauen wir uns eine Mview auf die Demo-Tabellen:

```
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;
```

# Constraints: Query Rewrite

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 |
-----------------------------------------------------------
```

# Constraints: Query Rewrite

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
```

# Constraints: Query Rewrite

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

- „typ“ ist der Primärschlüssel von „demo_typ“, d.h., jeder Datensatz in „demo“ passt zu maximal einem Datensatz in „demo_typ“.
- „typ“ in der Tabelle „demo“ hat eine Fremdschlüsselbeziehung zu „demo_typ“.
- „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.

# Constraint-Tuning: DEFERRABLE

- Seit Oracle 8.0
- Wird erst beim COMMIT geprüft
  - Dadurch Reihenfolge der Beladung von Tabellen mit Fremdschlüsselbeziehungen unerheblich
  - Geschwindigkeitsvorteil bei Massenbeladungen
- Bei Fehlern wird ROLLBACK ausgeführt
- Zweistufiges Prinzip:
  - Wird mit `deferrable initally [immediate | deferred]` angelegt
  - In der Session: `set constraint all deferred`

# Constraint-Tuning: DEFERRABLE

```
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.
```

# Constraint-Tuning: DEFERRABLE

**Achtung:** Mit dem deferrable FK geht nun die Table Elimination von oben 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 |
-----------------------------------------------
```

Außerdem: Wenn Deferrable Constraints vorliegen, können Operationen auf den betroffenen Tabellen nicht parallelisiert werden.

# Oracle Old Features

Weitere „Old Features”

# NVLst Du noch oder COALESCEt Du schon?

- Das gute, alte NVL()
  - Ist eine Oracle-spezifische Funktion
  - Vermutlich immer noch die beliebteste Methode zum Vergleichen und Ersetzen von NULLs
- Nachteile:
  - Kann nur genau einen Ausdruck mit einem anderen ersetzen; muß ansonsten kaskadiert werden
  - Der zweite Ausdruck wird *immer* ausgewertet, auch dann, wenn der erste Ausdruck bereits NOT NULL ist.

# NVL() vs. COALESCE()

Bsp.: 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:

```
CREATE OR REPLACE FUNCTION get_comm( job_id IN employees.job_id%TYPE )
RETURN NUMBER AS
BEGIN
  DBMS_OUTPUT.PUT_LINE( 'Aufruf get_comm()' );
  RETURN DECODE( job_id, 'SA_REP', 0.25, 0 );
END;
/

SELECT employee_id
     , NVL( commission_pct, get_comm( job_id )) comm
  FROM employees;

Aufruf get_comm()
Aufruf get_comm()
Aufruf get_comm()
… (viel mehr Aufrufe als NULLs in der Tabelle sind)
```

# COALESCE()

- Gibt es seit Oracle 9i
- ermöglicht die Auswertung von mehr als zwei Ausdrücken; der erste, der NOT NULL ist, wird zurückgeliefert.
- „Short Circuit”:
  - Funktion wird beendet, sobald der erste Nicht-NULL-Ausdruck gefunden wurde.
  - Entscheidender Geschwindigkeitsvorteil!
- COALESCE ist ANSI-SQL-konform.

# COALESCE()

Das Beispiel von oben in umgeschriebener Form:

```
SELECT employee_id
     , COALESCE( commission_pct, get_comm( job_id )) comm
  FROM employees;

Aufruf get_comm()
Aufruf get_comm()
(Nur so viele Aufrufe, wie NULLs in der Tabelle sind)
```

Ein vollständiger Umstieg von NVL auf COALESCE ist vielleicht Geschmackssache, bringt aber auf jeden Fall keine Nachteile.

# Materialisieren mit WITH

- Seit Oracle 9i
- Ideal für mehrfach verwendete Unterabfragen
  - Abfrage wird dann einmalig materialisiert und diese Menge dann wiederverwendet
  - Aufwendige Scans und Joins werden daher auch nur einmal statt mehrfach ausgeführt
  - Macht SQL-Code übersichtlicher
  - Spart außerdem noch Tipparbeit. :-)

# Materialisieren mit WITH

- Beispielszenario
  - Wir wollen aus einer Tabelle „depot_kunde“ Daten anzeigen, und zwar gefiltert über zwei Listen in den Tabellen „tmp_uid“ und „tmp_dep“.
  - Sind Schlüssel aus dem Datensatz *in einer der beiden Listen* enhalten, soll der Datensatz angezeigt werden.
  - Sind Schlüssel aus dem Datensatz *in beiden Listen* enhalten, soll der Datensatz *nicht* angezeigt werden.

# Materialisieren mit WITH

Aufbau des Testszenarios:

```
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
);

...
```

# Materialisieren mit WITH

```
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
```

Ein möglicher Lösungsansatz vor Oracle 9i:

```
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 );
```

# Materialisieren mit WITH

```
-----------------------------------------------
| 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
        600  redo size
```

# Materialisieren mit WITH

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 );
```

# Materialisieren mit WITH

```
-------------------------------------------------------------------
| 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  |
-------------------------------------------------------------------
Statistics
----------------------------------------------------------
          2  recursive calls
          8  db block gets
         37  consistent gets
          1  physical reads
        600  redo size
```

# Materialisieren mit WITH

Betrachten wir hier einmal auch die Prädikaten-Info näher:

```
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)
```

# Materialisieren mit WITH

- Caveats
  - Nicht immer automatische Materialisierung, z.B., wenn WITH-Block nur einmal referenziert wird. Aber auch bei weiteren Szenarien (und Bugs)
  - Hint /*+ MATERIALIZE */ innerhalb des WITH-Blocks erzwingt Materialisierung.
  - Optimizer geht gelegentlich nicht mehr alle Optimierungsansätze durch. Es kann dann gegenüber dem ursprünglichen SQL zu längeren Laufzeiten kommen.
  - → Testen, testen und nochmals testen!

# Oracle Old Features

?!

# Oracle Old Features

Herzlichen Dank für Ihre Aufmerksamkeit!

Uwe M. Küchler, Valentia GmbH

oraculix.wordpress.com

www.valentia.eu