



PL/SQL-Neuerungen der Versionen 10g und 11g

DOAG Konferenz Nürnberg 2008

Dr. Hildegard Asenbauer

MuniQSoft GmbH

- ◆ **Gegründet 1998**
- ◆ **Tätigkeitsbereiche:**
 - ▶ **Oracle Schulungen**
 - ▶ **Oracle IT Consulting & Services**
 - ▶ **Software Lösungen**
 - ▶ **Oracle Lizenzen**



MuniQSoft GmbH
Grünwalder Weg 13 a
D-82008 Unterhaching b. München
www.muniqsoft.de
+49(0)89-67909040

Übersicht

- ◆ Bulk DML
- ◆ Result Cache
- ◆ Performance-Optimierungen

Performance

-
- ◆ Erweiterungen zu Nested Tables
 - ◆ Compiler Warnungen
 - ◆ Reguläre Ausdrücke

Übersicht (f)

- ◆ **Bedingte Kompilierung**
- ◆ **Trigger-Erweiterungen in 11g**
- ◆ **Weitere Neuerungen in 11g**
- ◆ **Interessante Package-Neuerungen in 10g**
 - ▶ **DBMS_UTILITY**
 - ▶ **DBMS_SCHEDULER**
 - ▶ **UTL_MAIL**
 - ▶ **DBMS_DDL**

Erweiterungen zu Bulk DML

Übersicht

◆ Syntax:

```
▶ FORALL index IN x ..y [SAVE EXCEPTIONS]  
    sql_statement;  -- DML-Befehl
```

◆ Performance-Steigerung

◆ Verwendung von Collections zwingend

◆ nur skalare Datentypen

Historie

◆ 9i: Erweiterung um `SAVE EXCEPTIONS`:

- ▶ Exception Handler: `SQL%BULK_EXCEPTIONS`

◆ 9.2: Record-Datentyp erlaubt, aber kein Zugriff auf einzelne Felder:

- ▶

```
FORALL i IN v_rec.FIRST..v_rec.LAST
    INSERT INTO tab VALUES v_rec(i);
```
- ▶

```
FORALL i IN v_rec.FIRST..v_rec.LAST
    UPDATE tab SET ROW = v_rec(i)
    WHERE col1 = v_col1(i);
```

10 g: Array mit Lücken

- ◆ ab Version 10g können auch Arrays mit Lücken verarbeitet werden
- ◆ Optional kann der zu verarbeitende Index-Bereich eingegrenzt werden
- ◆ Syntax:
 - ▶ `FORALL index IN INDICES OF <collection>
[BETWEEN x AND y] [SAVE EXCEPTIONS]
sql_statement;`

bsp_forall1.sql

10g: Verwendung einzelner Indices

- ◆ Ebenfalls ab Version 10g kann genau eingeschränkt werden, welche Indices verwendet werden sollen:

- ◆ **Syntax:**

- ▶ `FORALL index IN VALUES OF <index_collection>`
`[SAVE EXCEPTIONS]`
`sql_statement;`

- ▶ `index_collection` muss eine Collection vom Typ `BINARY_INTEGER` oder `PLS_INTEGER` sein

11g: Arbeiten mit Record-Collections

- ◆ **Endlich** kann im DML-Befehl auf einzelne Felder des Records zugegriffen werden!

- ◆ Damit uneingeschränkt für alle DML-Befehle nutzbar:

```
▶ FORALL i IN v_rec.FIRST..v_rec.LAST
    UPDATE tab
        SET col1 = v_rec(i).field1
        WHERE col2 = v_rec(i).field2;
```

```
▶ FORALL i IN v_rec.FIRST..v_rec.LAST
    DELETE FROM tab
        WHERE col1 = v_rec(i).field1
```

DML Error Logging

- ◆ **Neues SQL-Feature ab 10.2**
- ◆ **Alternative zu SAVE EXCEPTIONS**
- ◆ **DML-Fehler werden protokolliert, führen aber nicht zu Abbruch**
- ◆ **Vorteil:**
 - ▶ **Genauere Info zu fehlerhaften Datensätzen**
- ◆ **Nachteil:**
 - ▶ **Kein Hinweis auf Fehler im Prozedurablauf**

DML Error Logging(f)

◆ Unterschied im Verhalten:

- ▶ **SAVE EXCEPTIONS: 1 Fehler / Statement**
- ▶ **LOG ERRORS: 1 Eintrag / Zeile**

◆ Logging-Tabelle:

- ▶ **DBMS_ERRLOG.create_error_log**

◆ DML-Befehl:

- ▶ `LOG ERRORS INTO <Logging-Tabelle>
REJECT LIMIT <Zahl> | UNLIMITED`

dbms_errlog.sql

Result Cache in 11g

Ausgangslage

- ◆ **Wiederholte Ausführungen von Selects und Funktionen kosten Zeit**
- ◆ **Selects: Statische Tabellen in Arrays vorhalten (Package-Variablen)**
 - ▶ **Vorteil: schnellerer Zugriff**
 - ▶ **Nachteile:**
 - nur innerhalb der gleichen Session
 - Speicherverbrauch
 - Keine Invalidierung bei Änderungen
- ◆ **Funktionen: Returnwert in Variable speichern**

RESULT CACHE

- ◆ **“Everyone knows the fastest way to do something is – *to not do it*“ (Tom Kyte)**
- ◆ **Result Cache in SGA → Session-übergreifend**
- ◆ **Neuer Parameter `result_cache_max_size`**
 - ▶ **Muss > 0 sein**
 - ▶ **Default abhängig von anderen init.ora-Parametern (`memory_target` / `sga_target` / `shared_pool_size`)**

RESULT CACHE in SQL

- ◆ Hint `result_cache`:

- ▶ `SELECT /*+ result_cache */ ...`

- ◆ Parameter `result_cache_mode` = FORCE

- ▶ Es wird immer versucht, den Result Cache zu nutzen

- ▶ Default: MANUAL

- ◆ Cache-Verwendung ist im Ausführungsplan sichtbar

RESULT CACHE in PL/SQL

- ◆ Schlüsselwort **RESULT_CACHE** bei Funktionsdeklaration
 - ▶ Package-Funktionen: Header UND Body
- ◆ Bei Zugriff auf Tabellen: **RELIES_ON**
 - ▶ Package-Funktionen: Body
 - ▶ Cache wird bei Tabellenänderung automatisch invalidiert
 - ▶ ACHTUNG: KEIN Fehler, wenn Klausel fehlt!
- ◆ Funktion wird bei weiteren Aufrufen nicht ausgeführt, wenn sich Parameter-Werte nicht geändert haben

RESULT CACHE

◆ Gespeichert werden:

- ▶ **Input-Parameter**
- ▶ **Ergebnis**

◆ Invalidierung:

- ▶ **Cache wird für EIGENE Session während offener Transaktion nicht mehr genutzt**
- ▶ **Invalidierung erst bei COMMIT**
- ▶ **→ Lesekonsistenz ist gewährleistet**

Beispiel

```
CREATE OR REPLACE PACKAGE p_cache
AS
    FUNCTION f_dept_count(p_deptno in number)
    RETURN NUMBER RESULT_CACHE;
END;
/
CREATE OR REPLACE PACKAGE BODY p_cache
AS
    FUNCTION f_dept_count(p_deptno IN NUMBER)
    RETURN NUMBER RESULT_CACHE
    RELIES_ON(emp)
    IS
        v_count NUMBER;
    BEGIN
        SELECT COUNT(*) INTO v_count FROM emp
        WHERE deptno = p_deptno;
        RETURN v_count;
    END;
END;
/
```

bsp_result_cache2.sql

Tuning und Info

- ◆ Package **DBMS_RESULT_CACHE**
 - ▶ **BYPASS, INVALIDATE, FLUSH, STATUS...**

- ◆ **V\$-Views**
 - ▶ **V\$RESULT_CACHE_OBJECTS**
 - ▶ **V\$RESULT_CACHE_MEMORY**
 - ▶ **V\$RESULT_CACHE_DEPENDENCY**

- ◆ **Weiterer Parameter**
 - ▶ **result_cache_max_result**

Einschränkungen

- ◆ **Keine OUT-Parameter**
- ◆ **Keine Selects auf SYS-Tabellen und -Views**
- ◆ **Nur skalare Parameter**
- ◆ **Keine LOBs, Ref Cursor oder Objekte als Returntypen**
- ◆ **Nicht bei Invoker Rights**

Performance-Optimierungen

Neue Datentypen

◆ **BINARY_FLOAT**

- ▶ **“single precision“ 32 Bit Datentyp**
- ▶ **benötigt 5 Byte (inkl. Längenbyte)**

◆ **BINARY_DOUBLE**

- ▶ **“double precision“ 64 Bit Datentyp**
- ▶ **benötigt 9 Byte (inkl. Längenbyte)**

◆ **IEEE 754 – konform**

◆ **Vorteile bei rechenintensiven Anwendungen**

PLSQL_OPTIMIZE_LEVEL

- ◆ **PL/SQL-Compiler und PVM wurden mit 10g komplett überarbeitet**
 - ▶ **Höhere Ausführungsgeschwindigkeit**
- ◆ **Neuer Parameter **PLSQL_OPTIMIZE_LEVEL** in 10g**
 - ▶ **0: keine zusätzlichen Optimierungen, Verhalten wie 9i**
 - ▶ **1: Optimierungen wie Entfernung unnötiger Berechnungen**
 - ▶ **2 (Default): weitergehende Optimierungen**
 - **z.B. implizite Umwandlung von CURSOR FOR-Schleifen in BULK SELECT**
 - ▶ **3 (11g): weitergehend als 2**
 - **u.a. auch automatisches Inlining**

PLSQL_OPTIMIZE_LEVEL (f)

◆ Auf System- / Session-Ebene einstellbar

- ▶ ALTER SYSTEM
SET PLSQL_OPTIMIZE_LEVEL=1;
- ▶ ALTER SESSION
SET PLSQL_OPTIMIZE_LEVEL=1;

◆ Für einzelne Programmeinheit einstellbar

- ▶ ALTER PROCEDURE <my_proc> COMPILE
plsql_optimize_level=2;

Native Kompilierung

- ◆ Eingeführt in 9i, aber umständlich
- ◆ Folgende Parameter entfallen in 10g:
 - PLSQL_COMPILER_FLAGS (deprecated)
 - PLSQL_NATIVE_C_COMPILER
 - PLSQL_NATIVE_LINKER
 - PLSQL_NATIVE_MAKE_FILE_NAME
 - PLSQL_NATIVE_MAKE_UTILITY
- ◆ Stattdessen **automatische** Suche nach OS-spezifischem C-Compiler (GCC, VC++, .Net)
 - ▶ Datei \$ORACLE_HOME/plsql/spnc_commands

Native Kompilierung

◆ Geblieben ist:

- PLSQL_NATIVE_LIBRARY_DIR
- PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT (optional)

◆ Neu:

- ▶ **PLSQL_CODE_TYPE** (NATIVE oder INTERPRETED)

◆ Native Kompilierung einstellbar:

- ▶ Auf System- und Sessionebene

- ALTER SESSION SET PLSQL_CODE_TYPE = 'NATIVE' ;

- ▶ Auf Prozedurebene

- ALTER PROCEDURE <my_proc> COMPILE
PLSQL_CODE_TYPE = NATIVE;

Native Kompilierung 10g

◆ Akzeptanz der Anwender niedrig

- ▶ Sicherheitsbedenken
- ▶ Zusatzkosten

◆ => Änderungen in 11g

Native Kompilierung 11g

- ◆ Kein C-Compiler mehr nötig
- ◆ Kein externes Verzeichnis mehr nötig
- ◆ Speicherung im SYSTEM-Tablespace
- ◆ Einzig nötiger Parameter: **PLSQL_CODE_TYPE**
- ◆ Neuer Datentyp **SIMPLE_INTEGER**

`bsp_nativ.sql`

SIMPLE_INTEGER

- ◆ Subtyp von PLS_INTEGER
- ◆ Gleicher Wertebereich (-2.147.483.648 bis 2.147.483.647)
- ◆ NOT NULL Constraint
- ◆ Kein Überlauf, stattdessen Wrapping
- ◆ Nur bei NATIVE mit Vorteilen

`bsp_simple_int.sql`

SIMPLE_FLOAT / SIMPLE_DOUBLE

- ◆ Ebenfalls ab 11g
- ◆ Subtypen von BINARY_FLOAT bzw. BINARY_DOUBLE
- ◆ Gleicher Wertebereich wie Basistyp
- ◆ Einziger Unterschied zu Basistyp: NOT NULL Constraint
- ◆ Nur bei NATIVE mit Vorteilen

Erweiterungen zu Nested Tables

Allgemeines

- ◆ **Umfassende Syntax-Erweiterungen in 10g:**
 - ▶ **Vergleich auf Gleichheit oder Ungleichheit möglich**
 - ▶ **Neue MULTISSET-Operatoren**
 - ▶ **Neue Bedingungen**
 - ▶ **Neue Funktionen**

- ◆ **betreffen NUR Nested Tables**

- ◆ **Auch in SQL nutzbar**

Vergleich von Nested Tables

- ◆ **Vergleich auf Gleichheit oder Ungleichheit möglich mit:**
 - ▶ =
 - ▶ !=
 - ▶ IN

- ◆ **Anzahl und Inhalt der Variablen werden verglichen, nicht jedoch die Reihenfolge.**

Multiset-Operatoren

- ◆ **Alle Operanden müssen vom gleichen Typ sein**
- ◆ **Operatoren:**
 - ▶ **MULTISET EXCEPT [ALL|DISTINCT]**
 - ▶ **MULTISET INTERSECT [ALL|DISTINCT]**
 - ▶ **MULTISET UNION [ALL|DISTINCT]**

 - ▶ **ALL ist der Default: Doppelte Einträge werden nicht eliminiert**
 - ▶ **Bei Angabe von DISTINCT werden doppelte Einträge eliminiert**

Multiset-Operatoren

◆ MULTISSET EXCEPT

◆ liefert eine Nested Table mit Elementen zurück, die nur in der ersten, nicht aber in der zweiten Nested Table sind

◆ Syntax:

▶ `<ntable1> MULTISSET EXCEPT [ALL|DISTINCT] <ntable2>`

Multiset-Operatoren (ff)

◆ MULTISSET INTERSECT

◆ liefert eine Nested Table mit Elementen zurück, die nur in beiden Nested Tables enthalten sind

◆ Syntax:

▶ `<ntable1> MULTISSET INTERSECT [ALL|DISTINCT]
<ntable2>`

Multiset-Operatoren (ff)

◆ MULTISSET UNION

◆ liefert eine Nested Table mit den vereinigten Elementen beider Nested Tables

◆ Syntax:

▶ `<ntable1> MULTISSET UNION [ALL|DISTINCT] <ntable2>`

Multiset-Operatoren: Beispiel

```
◆ DECLARE
  TYPE ttable IS TABLE OF NUMBER;
  v1      ttable := ttable(1, 2, 3, 3);
  v2      ttable := ttable (3, 3, 1, 1, 4);
  result  ttable;
BEGIN
  result := v1 MULTISSET UNION v2; --(1,2,3,3,3,3,1,1,4)
  result := v1 MULTISSET UNION DISTINCT v2; -- (1, 2, 3, 4)
  result := v1 MULTISSET INTERSECT v2;      -- (1, 3, 3)
  result := v1 MULTISSET INTERSECT DISTINCT v2; -- (1, 3)
  result := v2 MULTISSET EXCEPT v1;      -- (1, 4)
  result := v2 MULTISSET EXCEPT DISTINCT v1; -- (4)
END;
```

bsp_multiset.sql

Bedingungen

◆ Bedingungen:

- ▶ MEMBER
- ▶ IS [NOT] EMPTY
- ▶ IS [NOT] A SET
- ▶ SUBMULTISET

Bedingungen

◆ MEMBER

◆ Syntax:

▶ **<ausdruck> MEMBER [OF] <ntable>**

▶ **Schlüsselwort OF ist optional**

◆ **Liefert TRUE, falls der Ausdruck einem Element der Nested Table entspricht**

◆ **Liefert NULL, falls *ausdruck* NULL ist oder *ntable* NULL ist**

Bedingungen (ff)

- ◆ **IS [NOT] EMPTY**
- ◆ **Syntax:**
 - ▶ **< ntable > IS [NOT] EMPTY**
- ◆ **IS EMPTY gibt TRUE zurück, falls Nested Table leer ist (ACHTUNG: leer, aber NICHT NULL!)**
- ◆ **NOT ist Umkehrung**

Bedingungen (ff)

- ◆ **IS [NOT] A SET**
- ◆ **Syntax:**
 - ▶ **< ntable > IS [NOT] A SET**
- ◆ **IS A SET** gibt TRUE zurück, falls Nested Table keine Duplikate enthält
- ◆ **NOT** ist Umkehrung

Bedingungen (ff)

◆ SUBMULTISET

◆ Syntax:

▶ **< ntable1 > [NOT] SUBMULTISET [OF] < ntable2 >**

▶ **Schlüsselwort OF ist optional**

◆ **SUBMULTISET gibt TRUE zurück, falls alle Elemente in *ntable1* auch in *ntable2* enthalten sind (→ Untermenge)**

◆ **NOT ist Umkehrung**

Funktionen

◆ **CARDINALITY(< ntable >)**

- ▶ **Datentyp NUMBER**
- ▶ **Gibt die Anzahl an Elementen in *ntable* zurück**
- ▶ **Falls *ntable* NULL ist, wird NULL zurückgegeben**

◆ **SET(< ntable >)**

- ▶ **Datentyp Nested Table**
- ▶ **Gibt Nested Table zurück, die keine Duplikate (mehr) enthält**

Bedingungen und Funktionen: Beispiel

```
◆ DECLARE
    TYPE ttable IS TABLE OF NUMBER;
    v1          ttable := ttable(1, 2, 3, 3);
    v2          ttable := ttable(1, 2, 3);
    result      BOOLEAN;
    v_count     NUMBER;
BEGIN
    result := v2 SUBMULTISET OF v1;           -- (TRUE)
    result := v1 NOT SUBMULTISET OF v2;      -- (TRUE)
    result := 2 MEMBER OF v1;                -- (TRUE)
    result := v1 IS A SET;                    -- (FALSE)
    result := v1 IS EMPTY;                    -- (FALSE)
    v_count := CARDINALITY(v1);               -- (4)
    v_count := CARDINALITY(SET(v1));          -- (3)
END;
```

bsp_compare.sql

Compiler Warnungen

Compiler Warnungen

- ◆ Eingeführt in 10g: “PLW-“
- ◆ Initialisierungsparameter **PLSQL_WARNINGS**
 - ▶ `ALTER SYSTEM SET PLSQL_WARNINGS = <settings>`
 - ▶ `ALTER SESSION SET PLSQL_WARNINGS = < settings>`
 - ▶ `ALTER PROCEDURE <prozedurname>`
`COMPILE PLSQL_WARNINGS = < settings>`
- ◆ Settings: **ENABLE | DISABLE | ERROR:<level>**

Compiler Warnungen (ff)

◆ Levels:

- ▶ SEVERE
- ▶ PERFORMANCE
- ▶ INFORMATIONAL
- ▶ ALL
- ▶ (*liste*)

◆ Beispiele:

```
ALTER PROCEDURE test
COMPILE PLSQL_WARNINGS='ENABLE:PERFORMANCE';

ALTER SESSION SET PLSQL_WARNINGS =
'ENABLE:(5000,5001,5002)', 'DISABLE:INFORMATIONAL';
```

Compiler Warnungen (ff)

◆ 10g: 11 Warnungen lt. Doku

▶ Funktion ohne Returnwert:

- PLW-05005: function *string* returns without value at line *string*
“Funktion *string* ergibt keinen Rückgabewert in Zeile *string*“

▶ Code, der nie ausgeführt wird:

- PLW-06002: Unreachable code
“Auf Code kann nicht zugegriffen werden“

◆ 11g: über 30, u.a.:

▶ Exception Handler ohne RAISE

- PLW-06009: procedure "string" OTHERS handler does not end in RAISE or RAISE_APPLICATION_ERROR

bsp_warnings.sql

Reguläre Ausdrücke

Reguläre Ausdrücke

- ◆ **Aus vielen Programmiersprachen schon länger bekannt**
- ◆ **Oracle hält sich an die Posix-Norm, die man nachlesen kann unter:**
 - ▶ `http://www.regular-expressions.info/`
- ◆ **In SQL und PL/SQL nutzbar**
- ◆ **Wesentlich mächtiger als LIKE bzw. entsprechende Funktionen, aber auch schwieriger**

Reguläre Ausdrücke in 10g

◆ REGEXP_LIKE

- ▶ analog zu LIKE (Ergebnis: True/False)

◆ REGEXP_INSTR

- ▶ analog zu INSTR (Ergebnis: Position des Vorkommens)

◆ REGEXP_SUBSTR

- ▶ analog zu SUBSTR (Ergebnis: Teilstring)

◆ REGEXP_REPLACE

- ▶ analog zu REPLACE (Ergebnis: String mit Ersetzungen)

REGEXP_LIKE

◆ Syntax:

▶ `REGEXP_LIKE(source_string, pattern
[, match_parameter])`

▶ pattern:

. jedes Zeichen bis auf Newline (Zeilenumbruch)
^ Beginn einer Zeile
\$ Ende der Zeile
***** gar nicht oder beliebig oft
? kein oder einmal, entspricht {0,1}
+ einmal oder beliebig oft, entspricht {1,}
[a|b] Entweder a oder b
[:alpha:], [:digit:], [:alnum:], [:print:] ...

REGEXP_LIKE (f)

► **match_parameter:**

- i** Groß-, und Kleinschreibung wird nicht berücksichtigt
- c** Groß-, und Kleinschreibung wird berücksichtigt
- n** Das Sonderzeichen "." kann auch für einen Zeilenumbruch stehen
- m** Die Zeichenkette wird als mehrzeilige Eingabe betrachtet.
^ und \$ können dann für jede Zeile angewendet werden.

Beispiele REGEXP_LIKE

◆ Alle Mitarbeiter, die entweder mit K,A oder S beginnen:

```
▶ SELECT * FROM emp  
WHERE regexp_like(ename, '^[KAS]');
```

◆ Mitarbeiter, deren Namen mit G oder H aufhört:

```
▶ SELECT * FROM emp  
WHERE regexp_like(ename, '[HG]$');
```

REGEXP_INSTR

- ▶ `REGEXP_INSTR(source_string, pattern
[, position [, occurrence
[, return_option [, match_parameter]]]])`
- ▶ **pattern**: wie bei `REGEXP_LIKE`
- ▶ **position**: Position, ab der gesucht wird
- ▶ **occurrence**: Das wievielte Vorkommen wird gesucht?
- ▶ **return_option**: Soll Position des Strings (0) oder Position danach (1) zurückgegeben werden?
- ▶ **match_parameter**: wie bei `REGEXP_LIKE`

Beispiele REGEXP_INSTR

◆ Suche Beginn des dritten Wortes:

```
▶ SELECT REGEXP_INSTR  
  ('DOAG Konferenz 2008 Nürnberg', '[^ ]+', 1, 3)  
FROM dual;
```

=> 17

◆ Suche drittes Vorkommen von Leerzeichen:

```
▶ SELECT REGEXP_INSTR  
  ('DOAG Konferenz 2008 Nürnberg', '[ ]+', 1, 3)  
FROM dual;
```

=> 21

REGEXP_SUBSTR

◆ Syntax:

- ▶ `REGEXP_SUBSTR(source_string, pattern [, position [, occurrence [, match_parameter]]])`
- ▶ **pattern:** wie bei `REGEXP_LIKE`
- ▶ **position:** Position, ab der gesucht wird
- ▶ **occurrence:** Das wievielte Vorkommen wird gesucht?
- ▶ **match_parameter:** wie bei `REGEXP_LIKE`

Beispiele REGEXP_SUBSTR

```
▶ SELECT REGEXP_SUBSTR(  
  'system/password@myhost:1521:ora9',  
  '[^:]+',  
  1,  
  3) as "SID name"  
FROM dual;
```

=> ora9

```
▶ SELECT  
REGEXP_SUBSTR('johann.maier@chaos.de', '[^.]+')  
AS Vorname,  
REGEXP_SUBSTR('johann.maier@chaos.de',  
  '[^.@]+', 1, 2) AS Nachname,  
REGEXP_SUBSTR('johann.maier@chaos.de', '[^@]+',  
  1, 2) As Email FROM dual;
```

=> johann maier chaos.de

REGEXP_REPLACE

◆ Syntax:

```
▶ REGEXP_REPLACE(source_string, pattern  
  [, replace_string [, position [, occurrence  
  [, match_parameter ] ] ] ] )
```

- ▶ **pattern:** wie bei REGEXP_LIKE
- ▶ **replace_string:** Ersetzungsstring
- ▶ **position:** Position, ab der gesucht wird
- ▶ **occurrence:** Das wievielte Vorkommen wird gesucht?
- ▶ **match_parameter:** wie bei REGEXP_LIKE

Beispiele REGEXP_REPLACE

- ▶ `SELECT
REGEXP_REPLACE('(089) 67909040','[^[:digit:]]')
FROM dual;
=> 08967909040`
- ▶ `SELECT REGEXP_REPLACE('(089) 67909040','[^0-9]')
FROM dual;
=> 089 67909040`
- ▶ `SELECT REGEXP_REPLACE('Hallo40 Welt','[^ a-z]') FROM
dual;
=> Hallo Welt`

11g: REGEXP_COUNT

◆ Liefert die Anzahl der Vorkommen des Suchmusters

◆ **Syntax:**

- ▶ `REGEXP_COUNT(source_string, pattern [, position [, match_parameter]])`
- ▶ **pattern:** wie bei `REGEXP_LIKE`
- ▶ **position:** Position, ab der gesucht wird
- ▶ **match_parameter:** wie bei `REGEXP_LIKE`

Beispiele REGEXP_COUNT

◆ Wie oft kommen Leerzeichen im String vor?

```
▶ SELECT REGEXP_COUNT  
   ('DOAG Konferenz 2008 Nürnberg', '[ ]+')  
   FROM dual;
```

=> 3

◆ Wie oft kommt ein bestimmter Teilstring vor?

```
▶ SELECT  
   REGEXP_COUNT('1234565432112374376123123', '123')  
   FROM dual;
```

=> 4

Bedingte Kompilierung

Generelles

- ◆ Mit Version 10.2 wurde die bedingte Kompilierung eingeführt
- ◆ Bedingungen werden zur Kompilierzeit ausgewertet, nicht zur Laufzeit
- ◆ "Prä-Prozessor-Direktiven"
- ◆ Syntax:

```
$IF ... $THEN  
[ $ELSE ]  
$END
```

Prä-Prozessor-Variablen

◆ Mit Prä-Prozessor-Variablen:

- ▶ Werden über "\$\$" angesprochen:

```
$IF $$<variable=wert> $THEN
```

- ▶ Werden über **plsql_ccflags** gesetzt:

```
ALTER SESSION SET plsql_ccflags = '<name>:<wert>';  
ALTER SYSTEM SET plsql_ccflags =  
'<name>:<wert>[, <name>:<wert>]';
```

→ **Neukompilierung nötig**

```
ALTER PROCEDURE <proc_name> COMPILE  
plsql_ccflags = '<name>:<wert>' [REUSE SETTINGS];
```

Beispiel 1

```
CREATE OR REPLACE PROCEDURE DEBUG_PROC IS
    v_dummy NUMBER;
BEGIN
    v_dummy := 20;
    .....
    $IF $$debug_flag $THEN
        DBMS_OUTPUT.PUT_LINE('Inhalt von v_dummy: '
                               || v_dummy);
    $END
    ...
END;
```

bedingte_kompilierung.sql

Package-Konstanten

- ◆ **Mit Package-Konstanten:**
 - ▶ **Werden normal angesprochen**

 - ▶ **Datentyp ist beschränkt auf:**
 - **BOOLEAN**
 - **VARCHAR2**
 - **PLS_INTEGER**

Beispiel 2

```
CREATE PACKAGE PAC_C IS
    is_debug CONSTANT BOOLEAN := FALSE;
END;
```

```
CREATE PROCEDURE DEBUG_PROC IS
    v_dummy NUMBER;
BEGIN
    v_dummy := 20;
    ...
    $IF PAC_C.is_debug $THEN
        DBMS_OUTPUT.PUT_LINE('Inhalt von v_dummy: '
                               || v_dummy);
    $END
END;
```

bedingte_kompilierung2.sql

DBMS_DB_VERSION

```
$IF DBMS_DB_VERSION.VER_LE_10 $THEN
    --Code für Version 10 oder älter
$ELSIF DBMS_DB_VERSION.VER_LE_11 $THEN
    --Versions 11 code
$ELSE
    --Version 12 oder größer Code
$END
```

◆ Package-Konstanten sind jeweiliger Version angepasst:

- ▶ VERSION
- ▶ RELEASE
- ▶ VER_LE_xxx

DBMS_PREPROCESSOR

◆ Macht Quellcode nach Prozessierung zugänglich

▶ Funktion zum Einlesen in Array

- `GET_POST_PROCESSED_SOURCE`
`RETURN source_lines_t`

▶ Prozedur zur Ausgabe über DBMS_OUTPUT

- `PRINT_POST_PROCESSED_SOURCE`

◆ Beide Unterprogramme sind überladen

Neuerungen zu Triggern in 11g

Compound Trigger

- ◆ EIN gemeinsamer Trigger für alle Level
- ◆ Globale Variablen und Konstanten möglich
- ◆ Optionale Abschnitte für jeden Level:
 - ▶ BEFORE STATEMENT
 - ▶ BEFORE EACH ROW
 - ▶ AFTER EACH ROW
 - ▶ AFTER STATEMENT
- ◆ => Vermeidung von **Mutating Table**-Problemen

Compound Trigger: Syntax

```
◆ CREATE [OR REPLACE] TRIGGER <name>
FOR <event>
COMPOUND TRIGGER

    BEFORE STATEMENT
    IS BEGIN ...
    END BEFORE STATEMENT;
    BEFORE EACH ROW
    IS BEGIN ...
    END BEFORE ROW;
    AFTER EACH ROW
    IS BEGIN ...
    END AFTER ROW;
    AFTER STATEMENT
    IS BEGIN ...
    END AFTER STATEMENT;
END;
```

Beispiel Compound Trigger

```
CREATE OR REPLACE TRIGGER check_sal
FOR UPDATE OF sal ON SCOTT.EMP
COMPOUND TRIGGER
    -- Globale Deklarationen:
    TYPE t_number    IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
    TYPE t_varchar  IS TABLE OF VARCHAR2(30)
                                INDEX BY PLS_INTEGER;
    TYPE t_avgsal    IS TABLE OF NUMBER INDEX BY VARCHAR2(30);

    v_avgsal t_number;
    v_job    t_varchar;
    v_avgJob t_avgsal;
```

Beispiel Compound Trigger (f)

BEFORE STATEMENT IS

BEGIN

```
SELECT AVG(sal), job
```

```
  BULK COLLECT INTO v_avgsal, v_job
```

```
  FROM emp e
```

```
  GROUP BY e.job;
```

```
FOR i IN 1..v_avgsal.COUNT
```

```
LOOP
```

```
  v_avgJob(v_job(i)) := v_avgsal(i);
```

```
END LOOP;
```

END BEFORE STATEMENT;

Beispiel Compound Trigger (f)

```
AFTER EACH ROW IS  
BEGIN  
    IF :NEW.sal > 1.5 * v_avgJob(:NEW.job)  
    THEN  
        RAISE_APPLICATION_ERROR(-20000,  
            'Verdienst ist zu weit über Durchschnitt');  
    END IF;  
END AFTER EACH ROW;  
END;  -- Ende des Triggers
```

compound_trigger.sql

Reihenfolge von Triggern

- ◆ Vor 11g war Reihenfolge der Ausführung bei gleichem Timing zufällig
- ◆ Ab 11g möglich: **FOLLOWS-Klausel**
 - ▶ CREATE OR REPLACE TRIGGER <trigger2>
<timing> <event>
FOLLOWS <trigger1>
- ◆ Referenzierter Trigger muss
 - ▶ Gleichen Level haben (Event, Tabelle)
 - ▶ Erfolgreich kompiliert sein
- ◆ Compound Trigger: **FOLLOWS** bezieht sich auf einzelne Abschnitte

Trigger DISABLED erstellen

◆ Weitere optionale Klausel:

```
▶ CREATE OR REPLACE TRIGGER <trigger2>  
  <timing> <event>  
  FOLLOWS <trigger1>  
  DISABLE | ENABLE
```

◆ Vorteil: Kompilierungsfehler führen nicht zum Absturz etwaiger Transaktionen

Weitere Neuerungen 11g

Inlining

- ◆ Inlining verhindert Overhead bei Aufruf von Subroutinen
- ◆ Neues Pragma: **INLINE**
 - ▶ `PRAGMA INLINE (prozedur, 'YES' | 'NO');`
 - ▶ `PLSQL_OPTIMIZE_LEVEL >= 2`
 - ▶ Position: unmittelbar vor Aufruf
- ◆ Automatisches Inlining: `PLSQL_OPTIMIZE_LEVEL = 3`
- ◆ Quellcode bleibt unverändert

Fine Grained Dependency Tracking

- ◆ **Granularität bei der Verfolgung von Abhängigkeiten ist nicht mehr das Objekt als Ganzes, sondern Element daraus**
- ◆ **Beispiele:**
 - ▶ **Bei einer Tabelle wird Spalte ergänzt**
 - ▶ **Bei einem Package wird Prozedur oder globale Variable ergänzt**
- ◆ **Objekte werden dadurch nicht mehr so leicht invalidiert**
- ◆ **ORA-4068er Fehler (→Package State) werden dadurch NICHT vermieden**

dependencies.sql

Dynamisches SQL

- ◆ **SQL-Befehle können bei jeder Form > 32 K sein**
 - ▶ CLOBs erlaubt
- ◆ **DBMS_SQL-Cursor kann in Ref Cursor verwandelt werden und umgekehrt:**
 - ▶ **DBMS_SQL.TO_REFCURSOR**
 - ▶ **DBMS_SQL.TO_CURSOR_NUMBER**
- ◆ **DBMS_SQL kann mit benutzerdefinierten Typen umgehen**
- ◆ **Cursor hijacking bei DBMS_SQL-Cursor per Default unterbunden (Bind und Execute mit gleicher user_id wie Parse, gleiche Rollen)**

CONTINUE-Anweisung

- ◆ **Bewirkt Verlassen des aktuellen Schleifendurchgangs und Beginn eines neuen Durchgangs**
- ◆ **Konditional mit WHEN oder ohne Bedingung**

▶ LOOP

.....

```
CONTINUE [WHEN <bedingung>];
```

.....

```
END LOOP;
```

Sequenzwerte

- ◆ **Direkter Zugriff möglich, kein SELECT auf DUAL mehr nötig**

```
▶ DECLARE
    my_val NUMBER;
BEGIN
    my_val := MY_SEQ.NEXTVAL;
    my_val := MY_SEQ.CURRVAL;
END;
```

Namentliche Notation in SQL

- ◆ Vor 11g war beim Aufruf einer Funktion aus SQL heraus nur positionale Notation erlaubt
- ◆ Mit 11g ist auch namentliche Notation oder Mixed möglich
- ◆ `SELECT my_function(p1 => 42) FROM DUAL;`

◆ PL/SCOPE

▶ ALTER SESSION

```
SET plscope_setting = 'IDENTIFIERS:ALL';
```

▶ Informationen über Benutzung von Identifiern werden gesammelt, Info wird im SYSAUX-Tablespace gespeichert

▶ View USER_IDENTIFIERS

▶ Soll Hyperlinks im SQL Developer ermöglichen

Interessante Package-Neuerungen 10g

DBMS_UTILITY

FORMAT_ERROR_BACKTRACE

◆ Bekannt:

- ▶ SQLERRM (Längenbegrenzung!)
- ▶ DBMS_UTILITY.FORMAT_ERROR_STACK

◆ Ab 10g:

- ▶ DBMS_UTILITY.**FORMAT_ERROR_BACKTRACE**
- ▶ Liefert **Zeilennummer**, wo Fehler auftrat
- ▶ Bei RAISE Reset der Zeilennummer

backtrace.sql

GET_CPU_TIME

- ◆ **Zeitmessung: zweimaliger Aufruf**
- ◆ **Delta: Zeit in Hundertstel Sekunden**
- ◆ **Bekannt:**
 - ▶ **DBMS_UTILITY.GET_TIME**
 - ▶ **Wie viel Zeit ist vergangen?**
- ◆ **Neu ab 10g:**
 - ▶ **DBMS_UTILITY.GET_CPU_TIME**
 - ▶ **Wie viel Zeit hat die CPU gebraucht?**

get_cpu_time.sql

Sonstige Neuerungen

◆ DBMS_UTILITY.GET_DEPENDENCY

- ▶ Ausgabe von Abhängigkeiten
- ▶ DBMS_OUTPUT muss freigeschaltet sein

◆ DBMS_UTILITY.VALIDATE

- ▶ Objekte validieren (falls dies möglich ist)

◆ DBMS_UTILITY.INVALIDATE

- ▶ Objekte explizit invalidieren

Neuer Job Scheduler: DBMS_SCHEDULER

SCHEDULER

- ◆ Ab 10g als Ersatz / Erweiterung zu DBMS_JOB:
 - ▶ Kann auch externe Programme aufrufen
 - ▶ Job-Ketten (ab 10.2)
 - ▶ Statt Zeitintervalle auch Event-Steuerung möglich (ab 10.2)
- ◆ Package **DBMS_SCHEDULER**
- ◆ Nicht mehr von `job_queue_processes` abhängig, nur Hintergrundprozess CJQ

Jobtypen

◆ Mögliche Jobtypen:

- ▶ **plsql_block** (entspricht in etwa DBMS_JOB)
- ▶ **stored_procedure**
- ▶ **executable** (Ausführung von Befehlen auf OS-Ebene)
- ▶ **ab Version 10.2: chain** (Jobketten)

DBMS_SCHEDULER

◆ Typ “executable“:

- ▶ unter Windows muss neuer Dienst OracleJobScheduler<SID> gestartet sein
- ▶ Keine direkte Angabe von Parametern bei Aufruf

◆ Event-basierte Jobs beruhen auf AQ

DBMS_SCHEDULER

◆ Berechtigungen:

- ▶ **CREATE JOB**
- ▶ **CREATE EXTERNAL JOB**

◆ Scheduler-Objekte:

- ▶ **Job: Aufgabe, die erledigt werden soll**
- ▶ **Program (optional): Programmeinheit**
- ▶ **Schedule (optional): Fahrplan**
- ▶ **Window, Window Group (optional): Zeitfenster**

Anlegen eines Jobs

- ◆ **Angabe von**
 - ▶ **Jobname**
 - ▶ **Jobtyp**
 - ▶ **Durchzuführende Aktion**
- ◆ **Mehrere überladene Prozeduren**
- ◆ **Job muss explizit aktiviert werden (ENABLED-Parameter)**
- ◆ **Angabe des Intervalls als**
 - ▶ **PL/SQL-Ausdruck (analog zu DBMS_JOB)**
 - ▶ **Kalendarischer Ausdruck**

Kalendarische Ausdrücke

- ◆ Angabe von **freq** zwingend
 - ▶ **FREQ = YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY, SECONDLY**
- ◆ Weitere mögliche Angaben (unvollständig):
 - ▶ **INTERVAL (1- 999)**
 - ▶ **BYMONTH (JAN-DEC oder 1-12)**
 - ▶ **BYDAY (MON – SUN)**
 - ▶ **BYHOUR (0- 23)**
 - ▶ **BYMINUTE (0 – 59)**
 - ▶ **BYSECOND (0 – 59)**

Beispiele zu Intervallen

◆ Jeden Montag:

▶ `FREQ=WEEKLY; BYDAY=MON;`

◆ Jeden zweiten Dienstag

▶ `FREQ=WEEKLY; INTERVAL=2; BYDAY=TUE;`

◆ Jeden 11. März

▶ `FREQ=YEARLY; BYMONTH=MAR; BYMONTHDAY=11;`

◆ Alle 10 Minuten

▶ `FREQ=MINUTELY; INTERVAL=10;`

CREATE Beispiel

```
BEGIN DBMS_SCHEDULER.CREATE_JOB(  
    job_name           => 'job1',  
    job_type           => 'stored_procedure',  
    job_action         => 'testproc',  
    start_date         => TRUNC(SYSDATE),  
    repeat_interval    => 'freq=DAILY;byhour=14',  
    enabled            => TRUE);  
  
END;
```

◆ **Achtung: Der Default bei ENABLED ist FALSE!**

Beispiel 2: Externes Programm

```
BEGIN
```

```
  DBMS_SCHEDULER.CREATE_JOB
```

```
    (job_name           => 'test',  
     job_type           => 'executable',  
     job_action         => '?\bin\sqlplus.exe',  
     start_date         => TRUNC(SYSDATE) + 1,  
     number_of_arguments => 2,  
     enabled            => FALSE);
```

```
  DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE  
    ('test', 1, 'scott/tiger');
```

```
  DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE  
    ('test', 2, '@D:\work\DOAG\Scheduler\doIt.sql');
```

```
  DBMS_SCHEDULER.ENABLE('test');
```

```
END;
```

Programs und Schedules

- ◆ **Dienen der Wiederverwertbarkeit**
- ◆ **Programs legen Jobtyp und durchzuführende Aktion fest**
- ◆ **Schedules legen Repeat-Intervalle fest**
- ◆ **Bei Schedules sind nur kalendarische Ausdrücke zulässig**

Beispiel

```
BEGIN
```

```
DBMS_SCHEDULER.CREATE_PROGRAM(
```

```
    program_name    => 'program1',
```

```
    program_type    => 'stored_procedure',
```

```
    program_action  => 'myproc',
```

```
    enabled         => TRUE);
```

```
DBMS_SCHEDULER.CREATE_SCHEDULE(
```

```
    schedule_name   => 'schedule1',
```

```
    repeat_interval => 'freq=monthly; byday=SAT;  
                    byhour=3');
```

```
END;
```

Beispiel (ff)

```
BEGIN
```

```
  DBMS_SCHEDULER.CREATE_JOB(
```

```
    job_name      => 'job3',
```

```
    program_name  => 'program1',
```

```
    schedule_name => 'schedule1',
```

```
    enabled       => TRUE);
```

```
END;
```

Job-Ketten (Chains) ab 10.2

- ◆ **Mit Job-Ketten können mehrere Jobs nacheinander ausgeführt werden**
- ◆ **Angabe von Bedingungen für einzelne Steps**
- ◆ **Zusätzliche Rechte nötig, die über DBMS_RULE_ADM vergeben werden:**
 - ▶ **DBMS_RULE_ADM.create_rule_set_obj**
 - ▶ **DBMS_RULE_ADM.create_evaluation_context_obj**
 - ▶ **DBMS_RULE_ADM.create_rule_obj**

Job-Ketten (Chains) ab 10.2 (f)

◆ **Folgende Schritte müssen ausgeführt werden:**

1. Erzeugen eines Chain-Objekts:

- `DBMS_SCHEDULER.CREATE_CHAIN`

2. Definieren der Programs für die einzelnen Schritte

- `DBMS_SCHEDULER.CREATE_PROGRAM`

3. Definieren der Chain-Schritte

- `DBMS_SCHEDULER.DEFINE_CHAIN_STEP`

Job-Ketten (Chains) ab 10.2 (f)

4. Regeln hinzufügen

- `DBMS_SCHEDULER.DEFINE_CHAIN_RULE`
- **Erster Step: condition => TRUE**
- **Weitere Steps: Verweis auf Stati aus vorherigen Steps**

5. Chain aktivieren

- `DBMS_SCHEDULER.ENABLE`

6. Job erzeugen, der auf das Chain-Objekt zeigt

- `DBMS_SCHEDULER.CREATE_JOB`
- `job_action`: Name der Chain

Neue VerwaltungsvIEWS

- ◆ DBA_SCHEDULER_PROGRAMS
- ◆ **DBA_SCHEDULER_JOBS**
- ◆ DBA_SCHEDULER_JOB_CLASSES
- ◆ DBA_SCHEDULER_WINDOWS
- ◆ DBA_SCHEDULER_PROGRAM_ARGS
- ◆ DBA_SCHEDULER_JOB_ARGS
- ◆ **DBA_SCHEDULER_JOB_LOG**
- ◆ **DBA_SCHEDULER_JOB_RUN_DETAILS**
- ◆ DBA_SCHEDULER_WINDOW_LOG
- ◆ DBA_SCHEDULER_WINDOW_DETAILS
- ◆ DBA_SCHEDULER_WINDOW_GROUPS
- ◆ DBA_SCHEDULER_WINDOWGROUP_MEMBERS
- ◆ DBA_SCHEDULER_SCHEDULES
- ◆ DBA_SCHEDULER_GLOBAL_ATTRIBUTE
- ◆ DBA_SCHEDULER_CHAINS (10.2)
- ◆ DBA_SCHEDULER_CHAIN_RULES (10.2)
- ◆ DBA_SCHEDULER_CHAIN_STEPS (10.2)
- ◆ DBA_SCHEDULER_RUNNING_CHAINS (10.2)

UTL_MAIL

UTL_MAIL

- ◆ Vereinfachte Mailversendung ab 10g
- ◆ Aus Sicherheitsgründen nicht standardmäßig installiert:
 - ▶ \$ORACLE_HOME\RDBMS\ADMIN\utlmail.sql
 - ▶ \$ORACLE_HOME\RDBMS\ADMIN\prvtmail.plb
- ◆ Initialisierungsparameter **SMTP_OUT_SERVER**, nicht im laufenden Betrieb änderbar
 - ▶ Servername [:port]
 - ▶ Mehr als ein Server möglich

Installation und Konfiguration

```
CONN / AS SYSDBA
```

```
@?\rdbms\admin\utlmail.sql
```

```
@?\rdbms\admin\prvtmail.plb
```

```
ALTER SYSTEM
```

```
SET SMTP_OUT_SERVER='myserver'
```

```
SCOPE=SPFILE;
```

```
SHUTDOWN IMMEDIATE;
```

```
STARTUP;
```

```
GRANT EXECUTE ON UTL_MAIL TO . . . .;
```

Unterprogramme

◆ SEND:

- ▶ Versenden einfacher Mails ohne Anhang

◆ SEND_ATTACH_RAW

- ▶ Versenden von Mails mit binärem Anhang

◆ SEND_ATTACH_VARCHAR2

- ▶ Versenden von Mails mit Textanhang

Parameter

◆ Alle Prozeduren:

- ▶ **sender**
- ▶ **recipients (ggf. durch Kommata getrennt)**
- ▶ **cc (ggf. durch Kommata getrennt)**
- ▶ **bcc (ggf. durch Kommata getrennt)**
- ▶ **subject**
- ▶ **message**
- ▶ **mime_type (Default: 'text/plain; charset=us-ascii')**
- ▶ **priority (Default: 3)**

Beispiel SEND

```
DECLARE
```

```
  v_msg VARCHAR2(32767);
```

```
  cr    VARCHAR2(2000);
```

```
BEGIN
```

```
  cr := CHR(13) || CHR(10);
```

```
  v_msg := 'Ab Version 10g gibt es ein neues Package.' || cr;
```

```
  v_msg := v_msg || 'Es heisst UTL_MAIL ';
```

```
  v_msg := v_msg || 'und dient der Versendung von E-Mails.';
```

```
UTL_MAIL.SEND
```

```
  (sender      => 'h.asenbauer@muniqsoft.de',
```

```
   recipients => 'info@muniqsoft.de',
```

```
   subject    => 'Neues Package',
```

```
   message    => v_msg);
```

```
END;
```

Attachments

◆ Anhänge:

- ▶ Beschränkt auf 32 K
- ▶ Werden in Form von Variablen übergeben

◆ Parameter:

- ▶ `attachment` (RAW bzw. VARCHAR2)
- ▶ `att_inline` (Default: TRUE)
- ▶ `att_mime_type` (Default: 'text/plain; charset=us-ascii' bzw. 'application/octet')
- ▶ `att_filename`

Beispiel mit Anhang

```
DECLARE
    v_file BFILE;
    v_buffer RAW(32767);
    v_len BINARY_INTEGER;
BEGIN
    v_file := BFILENAME('UTL_DIR', 'logo.gif');
    DBMS_LOB.FILEOPEN(v_file, DBMS_LOB.FILE_READONLY);
    v_len := DBMS_LOB.GETLENGTH(v_file);
    DBMS_LOB.READ(v_file, v_len, 1, v_buffer);

    UTL_MAIL.SEND_ATTACH_RAW(
        sender      => 'h.asenbauer@muniqsoft.de',
        recipients  => 'info@muniqsoft.de',
        subject     => 'Logo',
        message     => 'Versenden eines Bildes',
        attachment  => v_buffer,
        att_inline  => FALSE,
        att_filename => 'logo.gif' );

    DBMS_LOB.FILECLOSE(v_file);
END;
```

DBMS_DDL

Online Wrapping

◆ DBMS_DDL.CREATE_WRAPPED

- ▶ Erzeugt angegebenes Objekt in verschlüsselter Form

```
BEGIN
```

```
    DBMS_DDL.CREATE_WRAPPED ('CREATE OR REPLACE  
                             PROCEDURE proc1 AS BEGIN NULL; END;');
```

```
END;
```

◆ DBMS_DDL.WRAP RETURN VARCHAR2

- ▶ Gibt CREATE-Befehl in verschlüsselter Form zurück

```
SELECT DBMS_DDL.WRAP ('CREATE OR REPLACE  
PROCEDURE proc1 AS BEGIN NULL; END;')  
FROM DUAL;
```