Kapitel 2
SQL und PL/SQL

Folien zum Datenbankpraktikum
Wintersemester 2009/10 LMU München

unter Verwendung der Folien des Datenbankpraktikums aus dem Wintersemester 2007/08 von Dr. Matthias Schubert

Übersicht

Syntax einer SQL-Anfrage

logische Abarbeitungsreihenfolge

select [**distinct**] <Attributsliste, (arithm.) Ausdrücke, ...>	**5**
from <Relationenliste, Tupelvariablen>	**1**
[**where** <Bedingung>]	**2**
[**group by** <Attributsliste>	**3**
[**having** <Gruppen-Bedingung>]]	**4**
[**union [all]** / **intersect** / **minus select ...**]	**6**
[**order by** <Liste von Spaltennamen>]	**7**

Bedeutung (Semantik) einer SQL-Anfrage

1. Kreuzprodukt aller Relationen, die in der **from**-Klausel vorkommen
2. Selektion aller Tupel, die die **where**-Klausel erfüllen
3. Partitionierung der Tupel in Gruppen, so dass die Tupel einer Gruppe in den Gruppierungsattributen übereinstimmen
4. Selektion aller Gruppen, die die **having**-Klausel erfüllen
5. Auswertung der **select**-Klausel: Auswahl der angegebenen Attribute (Projektion) und Elimination von Duplikaten, falls **select distinct** angegeben ist
   - ohne group by: für jedes in Schritt 2 verbliebene Tupel wird ein Ergebnistupel erzeugt
   - mit group by: für jede in Schritt 4 verbliebene Gruppe wird ein Ergebnistupel erzeugt
6. Auswertung des zweiten **select**-Statements und Durchführung der Mengenoperation
7. Sortierung entsprechend der **order by**-Klausel

Bemerkung: Die tatsächliche Auswertung einer Anfrage läuft in der Regel nicht in der Reihenfolge dieser Schritte ab (Performanz), sie muss nur das gleiche Ergebnis liefern.

Einfache Anfragen

- einzelne Attribute einer Relation R, die eine bestimmte Bedingung erfüllen:

```
select * from R where a1 > a7
```

- Join:

```
select R1.a1, R1.a2 from R1, R2 where R1.a1 = R2.a7
```

- Bedingungen (Prädikate):
  - einfache Vergleiche: **< > =** ... , verknüpft durch **and**, **or**, **not**
  - Pattern matching: **like 'literals'** mit **%** für beliebigen String, **_** für beliebiges Zeichen
  - Duplikate sind möglich, Abhilfe: **select distinct**
  - Attributeindeutigkeit durch Voranstellung des Relationennamens

- Tupelvariable (zur Abkürzung oder zur Eindeutigkeit bei Self-Joins):

```
select r1.a1 from R r1, R r2 where r1.a1 > r2.a2
```


- Outer Join: Liefert alle Tupel des einfachen Joins *und* alle Zeilen der einen Relation, die mit keiner Zeile der anderen Relation 'matchen' (mit `null`-Werten in den Join-Attributen).

  Beispiel: `Teilnahme (PersNr, Projekt)`

  `Angestellter (PersNr, Name)`

  "Ermittle die Namen aller Angestellten zusammen mit der Bezeichnung des Projekts, an dem sie arbeiten."

```
select Name, Projekt from Angestellter, Teilnahme
where Angestellter.PersNr = Teilnahme.PersNr
```

  liefert nur Angestellte, die an Projekten mitarbeiten.

```
select Name, Projekt from Angestellter, Teilnahme
where Angestellter.PersNr = Teilnahme.PersNr(+)
```

  liefert alle Angestellten.

  Neuere Syntax:

```
select Name, Projekt
from Angestellter left outer join Teilnahme using (PersNr)
```

Select-Klausel

Mit den Attributen (Spalten) kann auch gerechnet werden:

- arithmetische Ausdrücke: **+**, **-**, **/**, *****
  - mit Konstanten
  - mit Attributen

- Aggregatsfunktionen: **min**, **max**, **sum**, **avg**, **count**, ...

  Beispiel: `Angestellter(PersNr, Gehalt, ...)`

  "Durchschnittsgehalt aller Angestellten."

```
select avg(Gehalt) from Angestellter
```


Subqueries

- In der **where**- oder **having**-Klausel können weitere **select**-Statements auftreten, sog. Subqueries (nested queries, geschachtelte Anfragen).
- **Einfache Subqueries** werden **einmal** ausgewertet, mit dem Ergebnis wird weitergerechnet.

  Beispiel: `Student (MatrNr, Name, Vorname)`

  `Teilnahme (MatrNr, Veranstaltung)`

  "Namen aller Studierenden, die an mindestens einer Vorlesung teilnehmen."

```
select distinct Name
from Student
where MatrNr in (select MatrNr from Teilnahme)
```

  Könnte auch mit Join gelöst werden:

```
select distinct Name
from Student s, Teilnahme t
where s.MatrNr = t.MatrNr
```

- **Korrelierte Subqueries:**
  - sind abhängig von der äußeren Query
  - werden für jedes Tupel ausgewertet, das in der äußeren Query bearbeitet wird

Das obige Beispiel kann auch mit korrelierter Subquery gelöst werden:

```
select Name
from Student s
where exists (select * from Teilnahme t where s.MatrNr =
                  t.MatrNr)
```

ein anderes Beispiel: `Angestellter(PersNr, Gehalt, Abteilung)`

"Alle Angestellten, die mehr verdienen als das Durchschnittsgehalt ihrer Abteilung."

```
select *
from Angestellter a
where a.Gehalt > (select avg(b.Gehalt) from Angestellter b
                  where a.Abteilung = b.Abteilung)
```


- **Subqueries in der from-Klausel:**

Beispiel:

```
select Name
from ( select Name, MatrNr from Student s, Teilnahme t
                        where s.MatrNr = t.MatrNr)
```

→ möglich, aber meist redundantes Statement

→ lässt sich ein besseres Beispiel finden?

Group by-Klausel

- Syntax: `group by <Gruppierungsattribute>`
- Wirkung: Mengen von Tupeln mit gleichen Werten in den angegebenen Attributen werden zu Gruppen zusammengefasst. Die Ergebnisrelation enthält ein Tupel für jede Gruppe.
- Die Attribute in der **select**-Klausel müssen Gruppierungsattribute sein. Andere Attribute dürfen nur in Aggregatsfunktionen vorkommen.
- Beispiel: *"Minimales und maximales Gehalt der Angestellten in jeder Abteilung."*

```
select Abteilung, min(Gehalt), max(Gehalt)
from Angestellter
group by Abteilung
```


Having-Klausel

- Syntax: `having <Bedingung>`
- Wirkung: Für jede Gruppe aus `group by` (oder ganze Ergebnismenge, falls kein `group by` angegeben) wird `<Bedingung>` geprüft und nur bei `true` in das Ergebnis aufgenommen.
- Beispiel: *"Minimales und maximales Gehalt der Angestellten von jeder Abteilung, in der das Durchschnittsgehalt unter EUR 1500 liegt."*

```
select Abteilung, min(Gehalt), max(Gehalt)
from Angestellter
group by Abteilung
having avg(Gehalt) < 1500
```

Order by-Klausel

- Syntax: `order by <Sortierungsattribute> [ASC | DESC]`
  (pro Sortierungsattribut)

  wobei ASC = ascending (aufsteigend), DESC = descending (absteigend)
- Wirkung: Ergebnistupel werden sortiert
- Beispiel:

```
select Name, Vorname
from Student
order by Name, Vorname
```

  Bemerkung: Statt Attributnamen kann auch die Position in der **select**-Liste angegeben werden (nützlich bei langen Ausdrücken):

```
select Name, Vorname
from Student
order by 1, 2
```


Mengenoperationen

- Syntax:
```
select ...
        { union [all] | intersect | minus }
    select ...
```
- Wirkung: Mengenoperation ∪ (mit/ohne Duplikatelimination), ∩, \ auf den Ergebnistupeln
- Beispiel:

```
select Name, Vorname
from Student
 union
select Name, Vorname
from Professor
```

Null-Werte

o Kein definierter Attributwert → verschiedene Interpretationsmöglichkeiten:

- Wert existiert nicht (nie für das Tupel)
- Wert ist derzeit nicht bekannt
- Wert ist bekannt, aber nicht verfügbar (z.B. aus Datenschutzgründen)

o Wie wird `null` behandelt?

- Wenn `null`-Werte an arithmetischen Operationen (`+, - ,/ , *`) beteiligt sind, wird das Ergebnis auch zu `null`.
- Wenn `null`-Werte an Vergleichsoperationen (<, >, =, ...) beteiligt sind, wird das Ergebnis `unknown`. Wenn das Gesamtergebnis `unknown` ist, dann ist dies äquivalent zu `false`.
- Für Test auf `null` die Operatoren `is null`, `is not null` verwenden!
- Bei Aggregierung werden `null`-Werte nicht gezählt!
- Bei Gruppierung, Duplikatelimination bilden `null`-Werte eine Gruppe.
- Bei Sortierung ist `null` größter Wert.

- Bei logischen Operatoren `and`, `or`, `not` → dreiwertige Logik

  z.B. für **and**:

and	true	false	unknown
true	true	false	unknown
false	false	false	false
unknown	unknown	false	unknown

Beispiel:
```
select Name,Vorname
from Student
where Vorname > 'S' and Vorname < 'T'
```

liefert keine `null`-Werte als Vornamen.

Übersicht

Was sind Views?

- Views sind virtuelle, abgeleitete Relationen.
- Inhalt wird definiert durch **select**-Anweisung.
- Views werden in der Regel nicht materialisiert abgespeichert, sondern nur ihre Definition.
- Verwendungszweck:
  - Datenschutz: der Zugriff auf bestimmte Zeilen oder Spalten einer Tabelle kann unterbunden werden.
  - Ausblenden unnötiger Informationen
- Syntax:
```
create view <name> (<attr1>, ..., <attrn>) as
select <arg1>, ..., <argn> from ...
```
- Viewdefinitionen in Textdateien (*.sql) editieren und mit `start` bzw. `@` in *SQL*PLUS* laden.

Übersicht

Motivation

- SQL bietet keine Rekursion (Iteration) → keine Berechnungsvollständigkeit!
- Beispiel:

  gegeben: Relation `Strasse(von, nach)`

  gesucht: ist "E" von "A" erreichbar ?

  Annahme: Hilfsrelation `erreichbar(ort)` ist verfügbar (anfangs leer)

  insert into erreichbar **values** (A);
  while (E ∉ erreichbar ∧ erreichbar wächst an) {
  insert into erreichbar
  select nach **from** strasse
  where von **in** (**select** * **from** erreichbar);
  }

  Ist mit SQL nicht ausdrückbar !

- Abhilfe:
  - Einbettung von SQL in eine Hostsprache (z.B. C, Java) → Embedded SQL, JDBC
  - Hersteller-spezifische SQL-Erweiterungen, z.B. PL/SQL in Oracle

'PL/SQL' (Oracle): Erweiterung von SQL um prozedurale Elemente

Eigenschaften:

- Tupelweise Verarbeitung
- Befehls-Blöcke
- Variablen-Deklarationen
- Konstanten-Deklarationen
- Cursor-Definitionen
- Prozedur- und Funktionsdefinitionen
- Kontroll-Befehle
- Zuweisungen
- Exception- und Fehlerbehandlung
- keine statischen DDL-Befehle

PL/SQL-Block:

declare | function | procedure

begin

exception

end;

/

falls noch eine Deklaration folgt

Variablendeklaration

Deklaration	Beispiel
varname type;	birthdate DATE; name VARCHAR(20) NOT NULL := 'Huber';
constname CONSTANT *type* := *wert*;	pi CONSTANT REAL := 3.14159;
Datentypen • alle SQL-Datentypen • Subtypen	CHAR, NUMBER, … z.B. bei NUMBER: FLOAT, DECIMAL, …
BOOLEAN	switch BOOLEAN NOT NULL := TRUE;
%TYPE	balance NUMBER(7,2); min_balance balance%TYPE; varname table.attr%TYPE;
%ROWTYPE	/* employee(name VARCHAR(20), salary NUMBER(7,2), dept NUMBER(3)); */ emp_rec employee%ROWTYPE;

Befehle

- SQL-Befehle (keine DDL-Befehle)
  - `insert, delete, update, select ... into ... from ...`
- Cursor-Befehle (siehe Abschnitt 2.4)
- Kontroll-Befehle
  - `if ... then .. else/elsif ... end if,`
  - `loop ... end loop, while ... loop ... end loop,`
  - `for` *`i`* `in` *`lb..ub`* `loop ... end loop,`
  - `exit, goto` *`label`*
- Zuweisungen
  - *`varname`* `:=` *`wert`*`;`
- Funktionen
  - alle in SQL zulässigen Funktionen ('+', '||', `TODATE(...)`, ...)

Ausnahmebehandlung (exception handling)

- Mechanismus zum Abfangen und Behandeln von Fehlersituationen
- Interne Fehlersituationen (in ORACLE):
  - Treten auf, wenn das PL/SQL-Programm eine ORACLE-Regel verletzt oder ein systemabhängiges Limit überschreitet
  - z.B. Division durch Null, Speicherfehler, etc.
  - Oracle-Fehler werden intern über eine Nummer (Fehlercode) identifiziert
  - Exceptions benötigen einen Namen → Zuordnung von Namen zu Fehlercodes notwendig
  - Namen für die häufigsten Fehler sind bereits vordefiniert, z.B.: `ZERO_DIVIDE`, `STORAGE_ERROR`, ...

- Externe Fehlersituationen:
  - werden vom Benutzer definiert
  - müssen deklariert werden
  - müssen explizit aktiviert werden

- Syntax:

```
DECLARE
   exc_name EXCEPTION;                    -- Deklaration
   ...
BEGIN
   ...
   IF ... THEN
      RAISE exc_name;                     -- Aktivierung
   END IF;
   ...
EXCEPTION
   WHEN exc_name THEN ...                 -- Behandlung
   WHEN zero_divide THEN ...
   WHEN OTHERS THEN ...
END;
```


- Einige Möglichkeiten der Ausnahmebehandlung:
  - Behebung des aufgetretenen Fehlers (wenn möglich)
  - Transaktion zurücksetzen (rollback)
  - Ggf. erneuter Versuch (z.B. bei Überlastung des Servers)
  - Fehlermeldung an den Anwender und Abbruch, z.B.:
    `raise_application_error(-20000,'Exception exc_name occurred');`

Beispiel (unter $DBPRAKT_HOME/Beispiele/PLSQL/Division_pl.sql)

```
SET SERVEROUTPUT ON                                  -- SQLPLUS-Kommando
DECLARE                                              -- Anonymer PL/SQL-Block (ohne Namen)
   res NUMBER;

   FUNCTION teilen (samp_num NUMBER) RETURN NUMBER IS      -- (AS)
     numerator NUMBER;
     denominator NUMBER;
     the_ratio NUMBER;
     lower_limit CONSTANT NUMBER := 0.72;
   BEGIN
     SELECT x, y INTO numerator, denominator         -- Schema result_table:
          FROM result_table                          -- Name       Null?    Type
          WHERE sample_id = samp_num;                -- ---------- -------- ---------
     the_ratio := numerator/denominator;             -- SAMPLE_ID NOT NULL NUMBER(3)
     IF the_ratio > lower_limit THEN                 -- X                  NUMBER
          RETURN the_ratio;                          -- Y                  NUMBER
     ELSE
          RETURN -1;
     END IF;
```

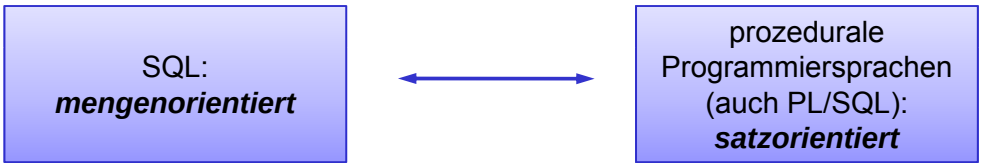


```
   EXCEPTION                                         -- gehört zur Funktion
     WHEN ZERO_DIVIDE THEN                           -- vordefinierte Ausnahme
          dbms_output.put('In EXCEPTION "ZERO_DIVIDE"');
          RETURN NULL;
     WHEN OTHERS THEN
          dbms_output.put('In EXCEPTION "OTHERS"');
          RETURN NULL;
   END teilen;

BEGIN                                    -- Hauptblock
   FOR i IN 130..133 LOOP                -- Inhalt der Relation result_table:
     res := teilen(i);                   -- SAMPLE_ID     X     Y
     dbms_output.put(i);                 -- ------------ ----- -----
     dbms_output.put(' ');               -- 130           70    87
     dbms_output.put(res);               -- 131           77    194
     dbms_output.new_line;               -- 132           73    0
   END LOOP;                             -- 133           81    98
END;                                     -- 133           54    76
/
SET SERVEROUTPUT OFF                     -- Welche Ausgabe liefert das Programm?
```

Übersicht

Problemstellung



- SQL-Anfrage liefert (in der Regel) Tupel**menge** als Ergebnis. Wie kann man damit in PL/SQL umgehen?

  → 2 Möglichkeiten:

  - 1-Tupel-Befehle für Anfragen, die max. 1 Tupel zurückliefern (z.B. `select ... into ...`)
  - Cursor: Elementweises Durchlaufen der Ergebnismenge und Einlesen jeweils eines Tupels in eine Variable

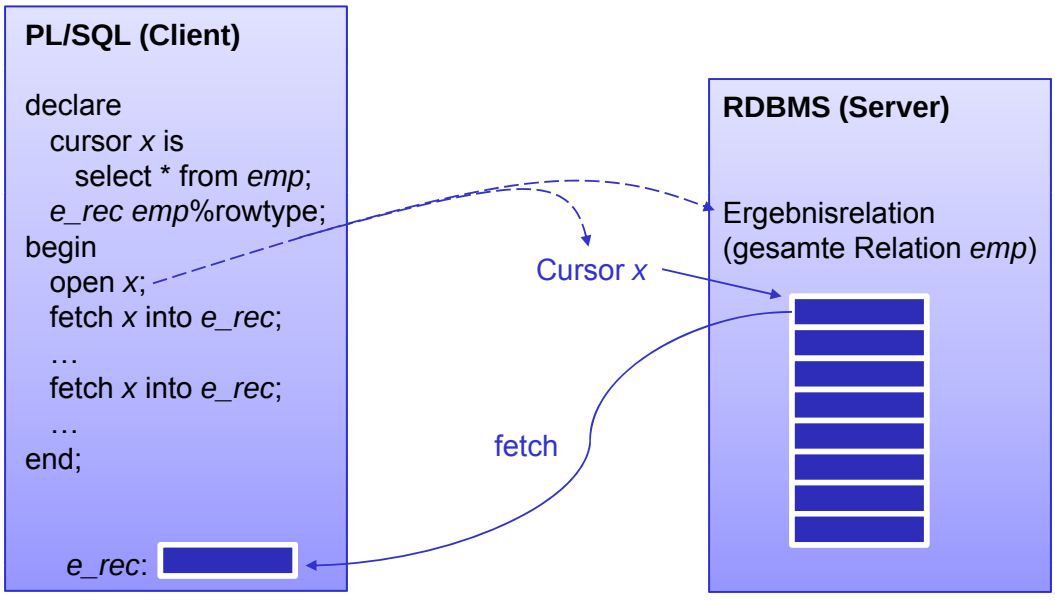
Cursor-Befehle

```
DECLARE
   CURSOR c1 IS                      -- Deklaration (Verbinden des Cursors
                                        mit einer Anfrage)
     SELECT n1 FROM data_table
     WHERE exper_num = 1;
   num1 data_table.n1%TYPE;          -- Attribut TYPE
BEGIN
   OPEN c1;                          -- Cursor öffnen (Auswerten der Anfrage
                                           Zeiger auf erstes Tupel setzen)
   LOOP                              -- Durchlaufen der Ergebnismenge
     FETCH c1 INTO num1;             -- aktuelles Tupel in Variable einlesen,
                                           Zeiger auf nächstes Tupel setzen
     EXIT WHEN c1%NOTFOUND;          -- Cursor-Attribut NOTFOUND
     ...
   END LOOP;
   CLOSE c1;                         -- Cursor schließen
END;
```


Schematische Darstellung

zur Cursor-Verwendung (gilt allgemein, nicht nur für die Verwendung innerhalb PL/SQL)

e_rec: Record-Variable, entspricht denau dem Tupel-Typ von *emp*



Noch ein Beispiel (unter $DBPRAKT_HOME/Beispiele/PLSQL/Cursor_pl.sql, für weitere Beispiele siehe Online-Doku zu PL/SQL)

```
DECLARE
   result temp.col1%TYPE;
   CURSOR c1(a NUMBER) IS                    -- Cursor mit Parameter
     SELECT n1, n2, n3 from data_table
     WHERE exper_num = a;
BEGIN
   FOR c1_rec IN c1(20) LOOP                 -- Cursor öffnen, Variable
                                             passenden Typs deklarieren,
                                             Ergebnismenge durchlaufen


     result := c1_rec.n2 / (c1_rec.n1 + c1_rec.n3);
     INSERT INTO temp VALUES (result, NULL, NULL);
   END LOOP;
END;
```


Übersicht

Stored Procedures

- PL/SQL-Prozeduren/Funktionen können als Objekte in der DB gespeichert werden.
- Befehl:
```
CREATE FUNCTION name (arg1 datatype1, ...)
    RETURN datatype AS ...

CREATE PROCEDURE name (arg1 [IN|OUT|IN OUT] datatype1, ...)
    AS ...
```
  bewirkt folgende Aktionen:
  - PL/SQL-Compiler übersetzt das Programm und prüft auf syntaktische und semantische Korrektheit.
  - Aufgetretene Fehler werden in das Data Dictionary eingetragen.
    → Abruf im SQL Developer: `SELECT * FROM USER_ERRORS`
    → Ausgabe der Beschreibung und der genauen Position der Fehler aller deklarierten Funktionen/Prozeduren/Trigger
  - Aufrufe von anderen PL/SQL-Programmen werden auf Zugriffsberechtigung überprüft.
  - Source Code und kompiliertes Programm werden in das Data Dictionary eintragen.
  - Rückmeldung an Benutzer
- Stored Procedures/Functions in Textfiles editieren!

- Abruf von Fehlermeldungen:
  - *SQL*PLUS:*
    ```
    show errors [procedure|function] <proc_name>
    ```
  - *Data Dictionary:*
    ```
    select * from user_errors where name = '<PROC_NAME>'
    ```
    (groß!)
- Aufruf von gespeicherten Prozeduren/Funktionen:
  - *aus PL/SQL:*
    ```
    [<schema_name>.]<procedure_name> (...);

    <var_name> := [<schema_name>.]<function_name> (...);
    ```
  - *aus SQL*PLUS:*
    ```
    execute [<schema_name>.]<procedure_name> (...)

    select [<schema_Name>.]<function_name> (...) from dual;
    ```
  Aufruf auch über andere Schnittstellen (z.B. ODBC/JDBC) möglich.

- Vorteile der Speicherung
  - reduzierte Kommunikation zwischen Client und Server ("network round-trips")
  - Programme bereits kompiliert → bessere Performance
  - gezielte Gewährung von Zugriffsrechten: das Ausführungsrecht für eine PL/SQL-Prozedur kann mit den Zugriffsrechten des Erstellers (*definer rights*, default) oder mit denen des Benutzers (*invoker rights*) ausgestattet werden
  - Teil der Anwendungsprogrammierung in DB zentralisiert
  - Verwaltung der Abhängigkeiten zwischen Programmen und anderen DB-Objekten durch DBMS

- Beispiel: (unter $DBPRAKT_HOME/Beispiele/PLSQL/Stored_pl.sql)

```
CREATE OR REPLACE FUNCTION inkrement (x number)
   RETURN number AS
 BEGIN
   return x+1;
END inkrement;
```

```
SQL> select inkrement(5) from dual;
```

wobei `dual` eine (1*1)-‚Dummy-Tabelle' ist, um das Ergebnis zu speichern

Übersicht

Packages

PL/SQL-Prozeduren/Funktionen können in Modulen (Packages) zusammengefasst werden.

Eigenschaften:

- Modulkonzept bietet mächtige Strukturierungsmöglichkeiten
- Package ist als Objekt in der DB gespeichert
- Trennung von Schnittstelle (*package specification*) und Implementierung (*package body*)
- Schnittstelle definiert nach außen sichtbare Funktions-, Prozedur-Köpfe, Typen, Cursor, Variablen, Konstanten
- Package-Name (und ggf. auch -Code) ist über Data Dictionary zugreifbar

Beispiel

```
CREATE OR REPLACE PACKAGE konvert AS              -- Definition der Schnittstelle
    PROCEDURE konvert_HS (anzahl NUMBER);
    PROCEDURE konvert_VA_WS0809 (anzahl NUMBER);
    FUNCTION konv_ok RETURN BOOLEAN;
END konvert;
```

```
CREATE OR REPLACE PACKAGE BODY konvert AS                    -- Implementierung
    <Var-, Cursor-Deklarationen, lokal zum Package Body>

    PROCEDURE konvert_HS (anzahl NUMBER) IS
      <Deklarationen, lokal zur Prozedur>
    BEGIN
      <Statements>
    END konvert_HS;

    PROCEDURE konvert_VA_WS0809 (anzahl NUMBER) IS ...
    END konvert_VA_WS0809;

    FUNCTION konv_ok RETURN BOOLEAN IS
      ok BOOLEAN;
    BEGIN ... RETURN ok;
    END konv_ok;

    PROCEDURE konv_loc1 ( ...) IS ... END konv_loc1;

    ...
END konvert;
```


Erstellen/Aufrufen von Packages

- Befehl `CREATE PACKAGE` bewirkt in etwa dieselben Aktionen wie `CREATE PROCEDURE/FUNCTION` (siehe Abschnitt ‘Stored Procedures’)
- Editieren in Textdatei
- Ausgabe zu Debugging-Zwecken mit vordefiniertem Package `dbms_output`
- Aufruf von Package-Prozeduren/Funktionen …
  - *aus SQL*PLUS:*
    `execute <schema_name>.<package_name>.<proc_name> (...)`
  - *aus PL/SQL:*
    `<schema_name>.<package_name>.<proc_name> (...);`

Übersicht

SQL-Standardisierungen

- herausgegeben durch ANSI und ISO
- Konformitätstests und Zertifizierung durch das NIST von 1980 bis 1996

o SQL86 (ANSI-SQL X3.135-1986 bzw. ISO 9075))

- Erster SQL-Standard

o SQL89 (ANSI-SQL X3.135-1989)

- Erweiterung von SQL 1986 um Embedded SQL + Referentielle Integrität

o SQL2 (auch SQL92 von 1992)

- Standardisierung von Erweiterungen, die die meisten Systeme schon bieten: Datumstypen, Funktionen, …
- verschiede Ebenen: Entry - Transitional - Intermediate - Full

o SQL3 (auch SQL:1999)

- Rekursion, Trigger, Autorisierung, ADT, Kapselung, Objektorientierung
- Anwendungen: Text, Geo, Zeit, Multimedia (SQL MM - Multimedia SQL) →LOBs

o SQL:2003

- Umgang mit XML, Größenbeschränkung von Anfrageergebnismengen (Window Functions), Sequenzen, Spalten mit automatisch generierten Werten