we focus on students

Datenbanken 1

Gespeicherte Funktionen und Prozeduren

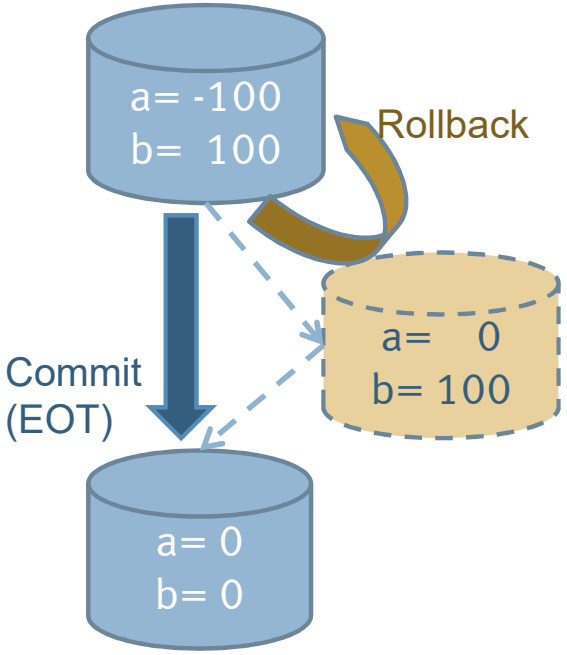
Fachhochschule Dortmund
University of Applied Sciences

Inhaltsübersicht

Transaktionen

Eine **Transaktion** ist eine inhaltlich zusammenhängende Menge von Datenbankoperationen, die ganz oder gar nicht ausgeführt werden.

Eine **Transaktion** überführt einen konsistenten Datenbankzustand in einen wiederum konsistenten Datenbankzustand.



Begin of Transaction (BOT)
Lese Kontostand a=-100
Lese Zahlungseingang b=100
Schreibe Kontostand a:=a +b
Schreibe Zahlungseingang b:=0
End of Transaction (EOT)

Commit *Rollback*

Syntax der SELECT-Anweisung

Was wird gesucht?

SELECT ‹Spalte$_1$›, ..., ‹Spalte$_n$›

Projektion (Festlegung der Ausgabespalten)

In welchen Tabellen?

FROM ‹Tabelle$_1$›, ..., **‹Tabelle$_m$›**

Join (Angabe der Tabellen und Verbundbedingung)

Auswahlbedingungen?

WHERE ‹Bedingung›

Selektionsbedingung (Auswahl der Tupel)
– optional

Gruppierung erforderlich?

GROUP BY ‹Spalte$_1$›, ..., ‹Spalte$_n$›

Gruppenbildung mit gleichen Werten
– optional

Gruppierungsbedingung?

HAVING <Bedingung>

Selektion von Gruppen
– optional unter group by

Sortierung?

ORDER BY ‹Attribut-Liste$_S$›

Sortierreihenfolge der Tupel in der Ergebnistabelle
– optional

INNER-JOIN Arten

- EQUI-JOIN: Überprüft auf Gleichheit von Attributen

```
SELECT a.Artikelnummer, Artikelname, Autor
FROM   Artikel a JOIN Warenkorb w
       ON a.Artikelnummer = w.Artikelnummer
```

- Vereinfachte Schreibweise

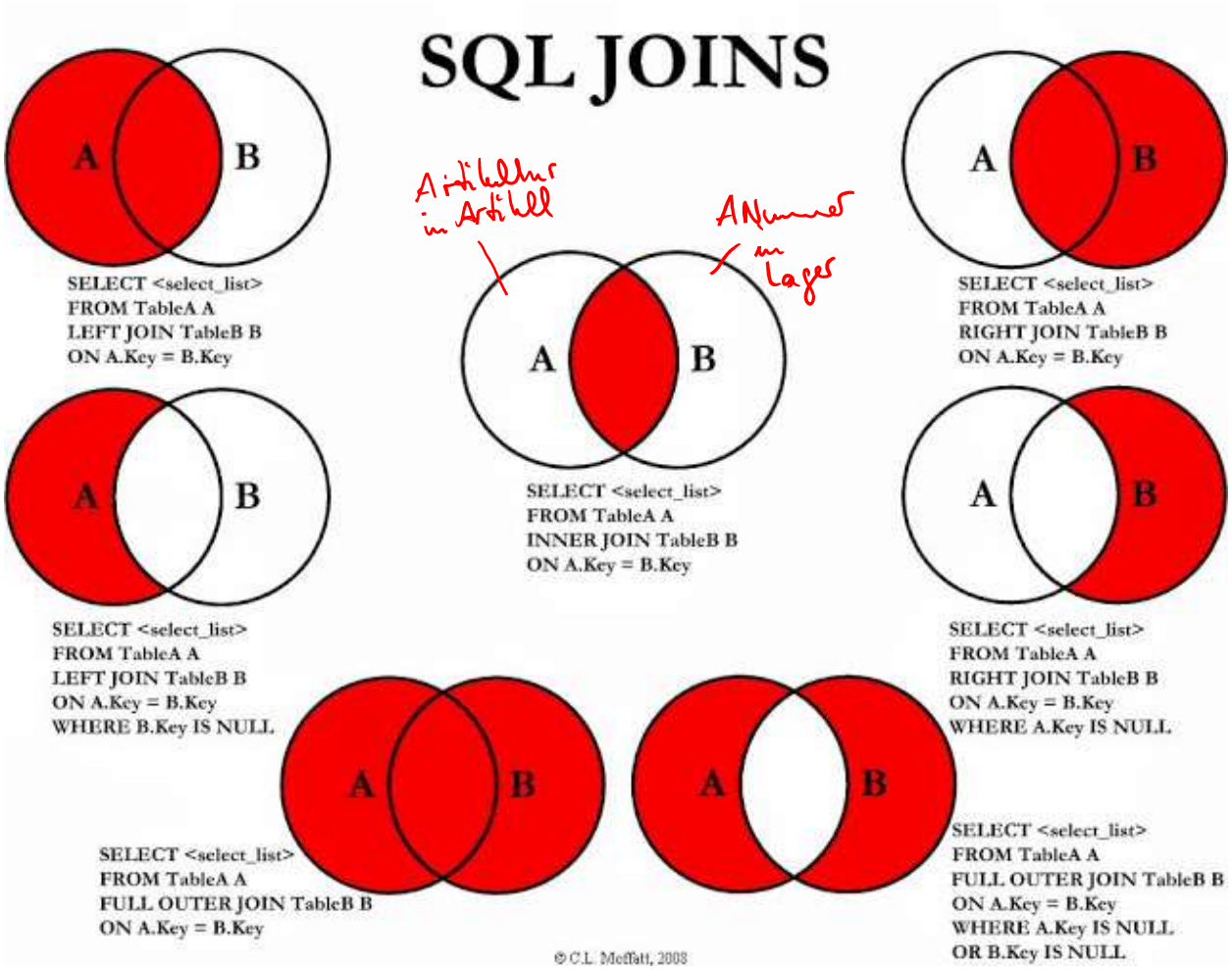
```
SELECT w.Artikelnummer, Artikelname, Autor
FROM   Artikel a JOIN Warenkorb w
       USING (Artikelnummer)
```

Attributliste möglich

- NATURAL JOIN: EQUI-JOIN über gleichbenannte Attribute

```
SELECT w.Artikelnummer, Artikelname, Autor
FROM   Artikel a Natural JOIN Warenkorb w
```

Übersicht Verbundoperationen



http://javarevisited.blogspot.de/2012/11/how-to-join-three-tables-in-sql-query-mysql-sqlserver.html#more

Die HAVING-Klausel

Auswahlbedingungen, welche sich auf das Ergebnis einer Gruppierung beziehen, müssen in der HAVING-Klausel angegeben werden.

SQL-Anfrage mit bedingter Gruppierung

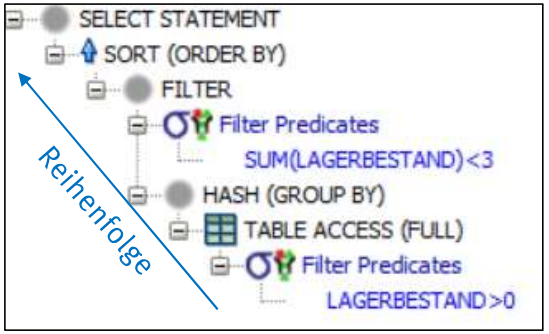
```
1  SELECT ANummer,Standort, SUM(Lagerbestand) AS Bestand
2  FROM Lager
3  WHERE Lagerbestand >0
4  GROUP BY ANummer, Standort
5  HAVING SUM(Lagerbestand) <3
6  ORDER BY Bestand DESC;
```

Abfrageergebnis

SQL | Alle Zeilen abgerufen:2 in 0,008 Sekunden

	ANUMMER	STANDORT	BESTAND
1	4811	INF	2
2	4814	MED	1

Interne Auswertungsreihenfolge



Unterabfragen – Verwendung und Ergebnismengen

Die Unterabfrage liefert

… einen einzelnen Wert

- **SELECT**-Clause
  - als Spaltenangabe
- **WHERE**-Clause
  - einer SELECT-Abfrage
  - einer DELETE-Anweisung
  - einer UPDATE-Anweisung
- **SET**-Clause
  - einer UPDATE-Anweisung

```
UPDATE Kunde
SET Ort = (SELECT Ort FROM Kunde
           WHERE Kundennummer=2310)
WHERE Kundennummer=8536
```

… eine Menge von Tupeln

- **FROM-Clause**
- **WHERE**-Clause
  - einer SELECT-Abfrage
  - einer DELETE-Anweisung
  - einer UPDATE-Anweisung

2 Vergleichsoperator - ALL

Ist die Ergebnismenge der Unterabfrage leer, dann liefert ein Vergleich mit dem ALL-Operator true. Diese Ergebnisse können durch eine zusätzliche Existenzbedingung ausgeschlossen werden.

Beispiel „Finde alle Artikel, die an jedem Standort mehr als viermal vorhanden sind [und hierfür auch Lagerplätze existieren].“

```
SELECT Artikelnummer, Artikelname
FROM Artikel a
WHERE 4 < ALL( SELECT Lagerbestand FROM Lager
                 WHERE  ANummer=a.Artikelnummer)
AND    EXISTS( SELECT Lagerbestand FROM Lager
                 WHERE  ANummer=a.Artikelnummer)
```

4812	Datenbanksysteme
4816	Anatomie-Atlas

Join oder Subquery?

„Finde alle Artikel, die **keinen** Lagerplatz zugewiesen bekommen haben“

- Subquery mit IN-Operator

```
SELECT Artikelnummer FROM Artikel
WHERE Artikelnummer  NOT IN
          (SELECT ANummer  FROM Lager WHERE ANummer IS NOT NULL )
```

- Subquery mit Exists-Operator

```
SELECT Artikelname, Autor, Ausgabe FROM  Artikel a
WHERE NOT EXISTS (SELECT ANummer From Lager
                        WHERE  ANummer=a.Artikelnummer)
```

- Mit Left-Join

```
SELECT DISTINCT Artikelname, Autor, Ausgabe, ANummer
FROM    Artikel a  LEFT JOIN  Lager l
          ON a.Artikelnummer = l.ANummer
WHERE ANummer IS  NULL
```

Join oder Subquery?

„Finde alle Artikel, zu denen an (mind.) einem Standort mehr als 2 Exemplare vorhanden sind“

- Unterabfrage mit Gruppierung

```
SELECT Artikelname, Autor, Ausgabe
FROM  Artikel a
WHERE 2 < ANY (SELECT SUM(Lagerbestand)  FROM Lager
                    WHERE ANummer=a.Artikelnummer
                    GROUP BY Standort)
```

- Verbundoperation mit bedingter Gruppierung

```
SELECT DISTINCT Artikelname, Autor, Ausgabe
FROM  Artikel a JOIN Lager l On a.Artikelnummer=l.ANummer
GROUP BY Artikelnummer, Standort
HAVING 2<SUM(Lagerbestand)
```

Strukturierung von Anfragen

Eine Unterabfrage in der FROM-Klausel stellt eine temporäre Benutzersicht (Inline View) dar. Diese ist nur für diese Anfrage verfügbar. Mit der WITH-Klausel kann die Unterabfrage zu Beginn der Abfrage deklariert werden.

Unstrukturierte Abfrage

```
SELECT   Lagernummer
FROM     Lager
WHERE    Standort = 'INF'
AND      Lagerbestand IS NULL AND ANummer IS NULL
```

Mit Unterabfrage strukturiert:

```
SELECT   frei.Lagernummer
FROM    (SELECT * FROM Lager
         WHERE  Lagerbestand IS NULL
         AND    ANummer IS NULL) frei
WHERE Standort = 'INF'
```

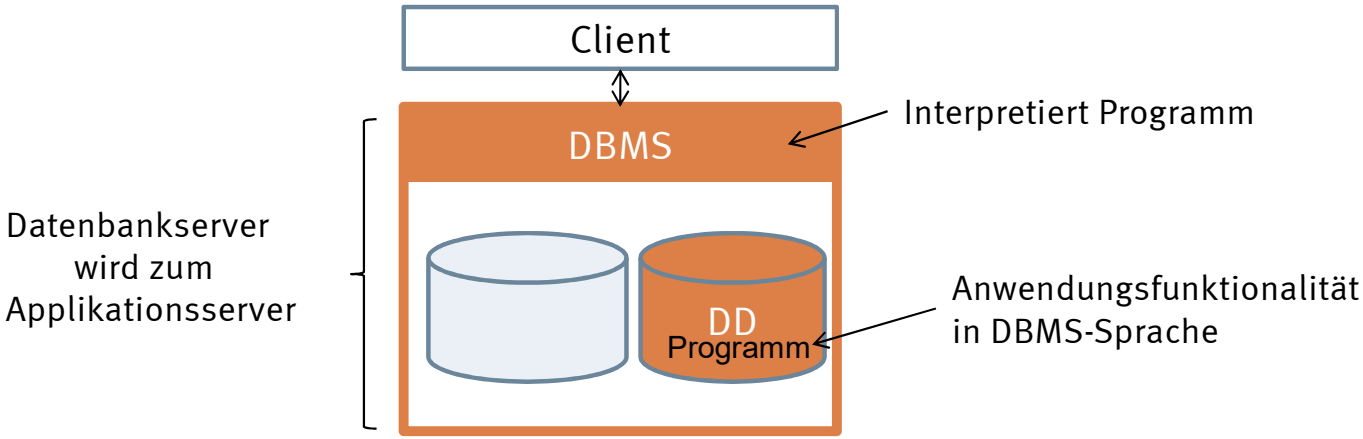
Mit WITH-Präfix:
(*Sub-Query Factoring*)

```
WITH frei AS (SELECT * FROM Lager
              WHERE  Lagerbestand IS NULL
              AND    ANummer IS NULL)
SELECT Lagernummer FROM frei
WHERE Standort = 'INF'
```

Inhaltsübersicht

Aktive Datenbank

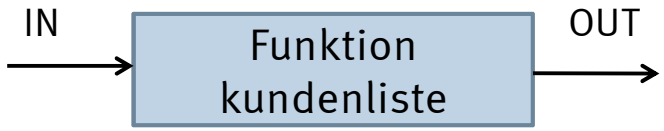
Bei einer **aktiven Datenbank** übernimmt das DBMS Anwendungsfunktionalitäten vom Client.



Programmierkonzepte (Datenbank)

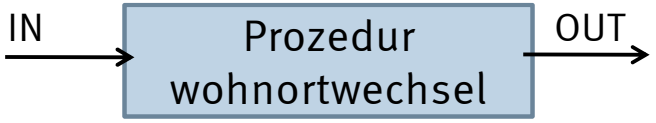
- Gespeicherte Funktionen (stored function)

```
SELECT kundenanrede(Kundennummer)
FROM    Kunde
```

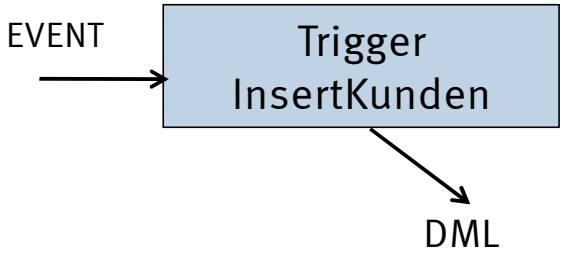


- Gespeicherte Prozeduren (stored procedures)

```
CALL kundenliste('Dortmund')
```



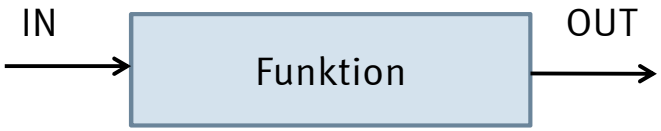
- Eventgesteuerte Prozeduren (Trigger)



Inhaltsübersicht

Konzept der Funktion

Konzept der Funktion



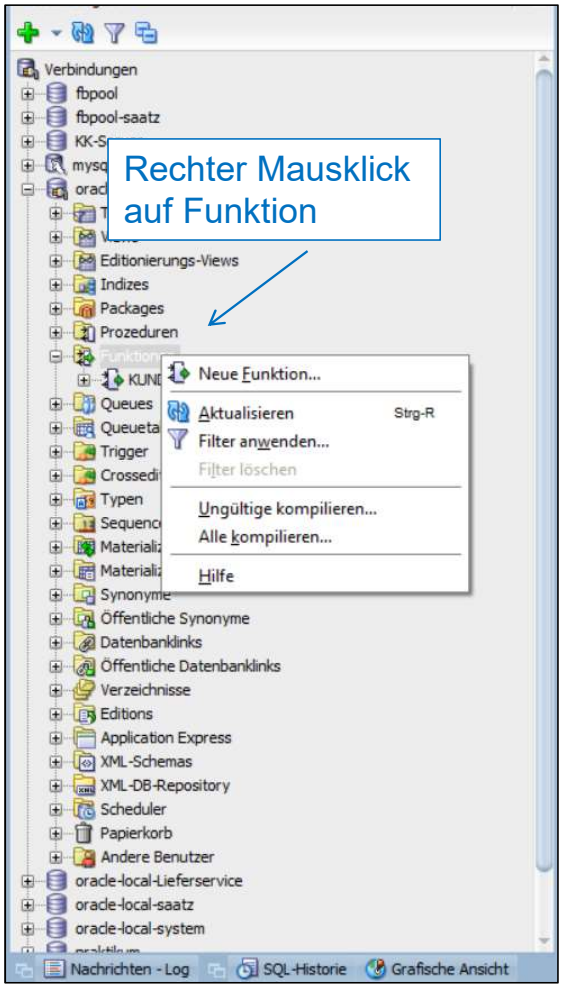
In Datenbanken

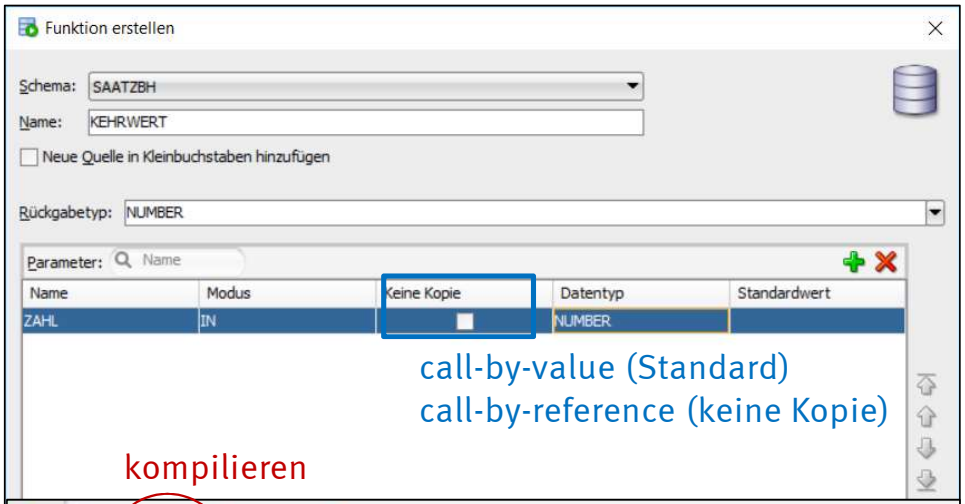
- Durch das DBMS bereitgestellte Funktionen
  - Spaltenfunktionen avg(*), sum(*), …
  - Datumsfunktionen (now(), extract(year from date)…)
  - Zeichenkettenfunktionen (Concat(), …)
  - Mathematische Funktionen (z.B. round(), sin(), …)

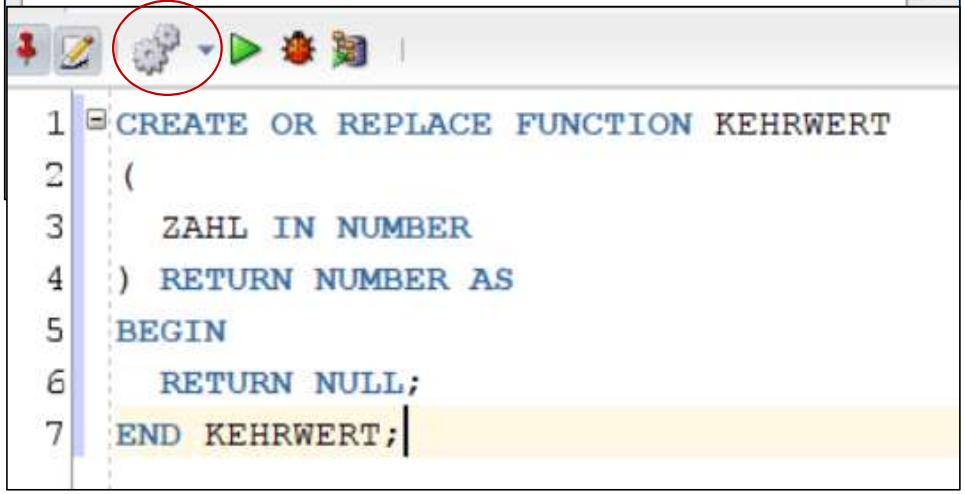
- Benutzerdefinierte Funktion

```
SELECT kundenanrede(Kundennummer)
FROM   Kunde
```

SQLDeveloper - Funktionen anlegen







Eine gespeicherte Funktion

Syntax

```
CREATE OR REPLACE FUNCTION  name(‹variablenliste›) RETURN ‹datentyp›
AS
   ‹variablendeklaration›
BEGIN
    ‹anweisungen›
    RETURN rueckgabewert;
EXCEPTION
    ‹ausnahmebehandlung›
END;
```

Beispiel

```
CREATE OR REPLACE FUNCTION kehrwert (zahl IN INTEGER) RETURN
NUMBER
AS
  rueckgabe NUMERIC( 9,8);
BEGIN
   rueckgabe:=1/zahl;
  RETURN rueckgabe;
END;
```

Hoffentlich ist zahl ‹› 0

Vorsicht Falle!

RETURN Numeric → Rückgabewert gerundet ohne Nachkommastellen
RETURN Number → Rückgabewert mit Nachkommastellen

Fallunterscheidung und Ausnahmebehandlung

Syntax

```
IF ‹bedingung› THEN     ‹anweisungen›
   [ELSIF ‹bedingung› THEN ‹anweisungen›]
   [ELSE ‹anweisungen›]
END IF;
```

```
CREATE FUNCTION name (...)
AS
      myfehlermeldung    EXCEPTION;
BEGIN
   IF ‹bedingung›
      THEN
           RAISE myfehlermeldung;
      END IF;
EXCEPTION
      WHEN myfehlermeldung
      THEN raise_application_error(-20500,'Mein Fehlertext');
END;
```

Fehler deklarieren

Fehler werfen

Fehler behandeln

Fehlernummer

Abbruch der Bearbeitung signalisieren

Inhaltsübersicht

Variablenzuweisung durch SQL-Anfragen

Die Wertzuweisung von lokalem Variablen kann durch SQL-Anfragen erfolgen. Dabei müssen die Variablenbezeichnungen sich von den Bezeichnungen der selektierten Attribute unterscheiden.

Deklaration lokaler Variablen

variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]

Wertzuweisung

var_name := *expr* | variable | konstante
SELECT *spalte*[,...] INTO *var_name*[,...] FROM tabelle

```
CREATE OR REPLACE FUNCTION kundenanrede2 (knr IN INTEGER)  RETURN VARCHAR
AS
    tmp_anrede     CHAR(4);
    tmp_nachname VARCHAR(30);
BEGIN
     SELECT anrede, nachname INTO tmp_anrede, tmp_nachname
     FROM   Kunde
     WHERE Kundennummer=knr;
     RETURN kundenanrede(tmp_anrede, tmp_nachname);
END;
```

Behandlung mengenwertiger SQL-Anfragen

- Bisher:
  - Variablen werden Werte zugewiesen durch Anfragen, die ein Tupel zurück liefern:

```
SELECT nachname INTO name
FROM   Kunde
WHERE  Kundennummer=2310;
```

- Jetzt:
  - Auswertung von mengenwertiger Anfragen im DB-Programm
  - Beispiel: Erstellung einer Funktion, die eine Kundenliste erstellt

```
SELECT Kundenliste() FROM dual;
```

```
Meitner:Einstein:Curie:Dekanat Informatik:Meier:
```

  - Lösungsidee
    - Die einzelnen Tupel der Ergebnismenge werden nacheinander in einer Schleife durchlaufen und verarbeitet

Cursor-Konzept

- DBMS

Satzanforderung (fetch)

Cursor liefert ein Ergebnistupel

```
SELECT Kundennummer, Nachname, Anrede
FROM Kunde
```

next()

	KUNDENNUMMER	NACHNAME		ANREDE
1	2310	Meitner	...	Frau
2	7562	Einstein	...	Herr

- Cursor

	Metadata	Spalte 1	Spalte 2	Spalte 3
Initial		Kundennummer	Nachname	Anrede
next()				
next()				
next()				

Cursor für mengenwertige Abfragen

Ein Cursor ist ein Zeiger auf ein Tupel der Ergebnismenge einer SQL-Anfrage. Aus dem Cursor werden solange Tupel für Tupel ausgelesen, bis keine weiteren Tupel im Cursor mehr vorhanden sind.

Definition eines Cursors

```
DECLARE CURSOR kundencursor
    IS    SELECT Nachname FROM Kunde;
```

Öffnen des Cursors (Berechnen der Ergebnismenge)

```
OPEN kundencursor;
```

Zugriff auf Tupel (i.d.R. in einer Schleife)

```
FETCH kundencursor INTO ‹variable›
```

Schließen des Cursors, Freigabe der Tupelmenge.

Bei Oracle erfolgt das Öffnen und Schließen des Cursors implizit bpsw. in einer For-Schleife

```
CLOSE kundencursor;
```

Funktion Kundenliste mit Cursor

Bei Oracle erfolgt das Öffnen und Schließen des Cursors implizit in der For-Schleife

```
CREATE OR REPLACE FUNCTION KUNDENLISTE
RETURN VARCHAR2
AS
  kundenname varchar(30);
  liste      varchar(210):= '';
  CURSOR kundencursor IS
     SELECT nachname FROM Kunde;
BEGIN
   FOR k in kundencursor LOOP
      liste := liste||TRIM(k.nachname)||':';
   END LOOP;
   RETURN liste;
END KUNDENLISTE;
```

Cursor deklarieren

Durchlaufen der Ergebnismenge

Implizites Schließen des Cursors

```
Meitner:Einstein:Curie:Dekanat Informatik:Meier:
```

Cursor-Status

Den Status eines Cursors beschreiben seine vier Attribute Found, NotFound, Rowcount und isOpen. Mit CURRENT OF wird auf das aktuelle Tupel verwiesen.

%FOUND

- Gibt an, ob der letzte FETCH - Befehl einen Satz gefunden hat => TRUE
- Vor dem ersten Fetch NULL

%NOTFOUND

- Gibt an, ob der letzte FETCH - Befehl einen Satz gefunden hat => FALSE
- Vor dem ersten Fetch NULL

%ROWCOUNT

- Liefert die Anzahl der mit FETCH gelesenen Zeilen
- Vor dem ersten FETCH auf 0

%ISOPEN

- Gibt an, ob ein Cursor geöffnet ist

Beispiel:

```
liste := liste||TRIM(k.nachname)||':'||kundencursor%rowcount;
```

```
1Meitner:1Einstein:2Curie:3Dekanat Informatik:4
```

Kontrollstrukturen – Zählschleifen

- FOR-Schleife
  - Syntax

```
FOR ‹var› IN [REVERSE] von … bis LOOP
        [exit [when ‹bedingung›]]
END LOOP
```

  - Beispiele

```
FOR i IN  1..10 LOOP
      --  tue was
END LOOP;
```

```
FOR yyyy IN (SELECT Preis FROM Artikel) LOOP
   -- tue was
END LOOP;
```

Kontrollstrukturen - Schleifen

Bei der WHILE-Schleife wird erst die Abbruchbedingung geprüft, bevor die Anweisungen ausgeführt werden. Bei der REPEAT-Schleife werden die Anweisungen zuerst ausgeführt, bevor die Abbruchbedingung geprüft wird.

- WHILE-Schleife

```
WHILE ‹bedingung› LOOP
        ‹anweisungen›
END LOOP
```

```
WHILE  i < 10 LOOP
        i:=i+1;
END LOOP;
```

- REPEAT-Schleife

```
LOOP
      ‹anweisungen›
      EXIT [WHEN ‹bedingung› ];
END LOOP;
```

```
LOOP
      i:= i+1;
      EXIT WHEN i=10;
END LOOP;
```

Inhaltsübersicht

Gespeicherte Prozeduren - Beispiel

Syntax

```
CREATE PROCEDURE ‹procedure_name›
   [(‹argument1›, ...) ]
{IS | AS}
  [‹Deklarationen lokaler variablen›]
BEGIN
   ‹ausfuehrbare anweisungen›

[EXCEPTION ‹ausnahmebehandlung›]

END [‹procedure_name›]
```

Definition

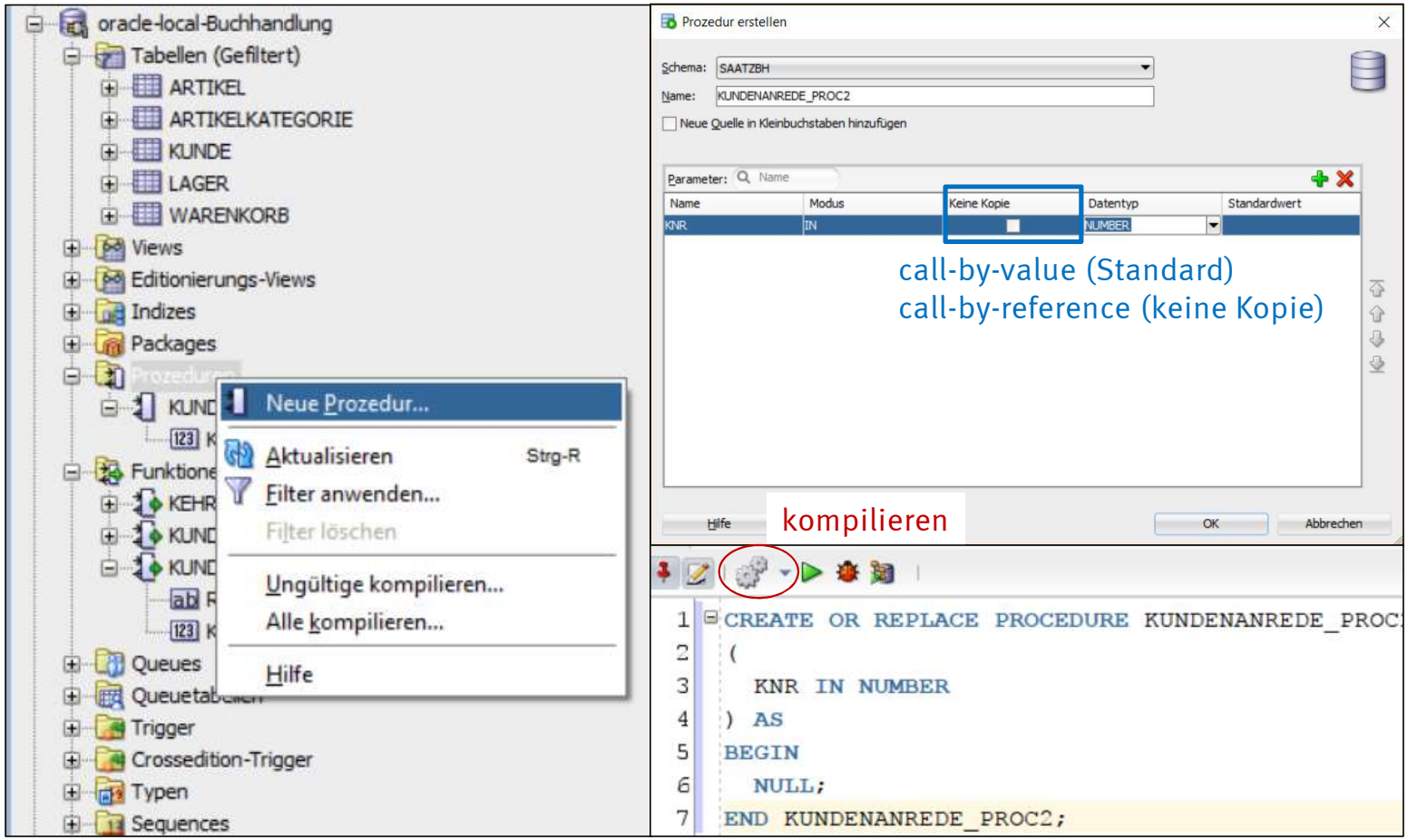
```
CREATE OR REPLACE PROCEDURE kundenanrede_proc (knr IN INTEGER)
IS
     anrede VARCHAR(60);
BEGIN
    SELECT Kundenanrede2(knr) INTO anrede  FROM dual;
    dbms_output.put_line(anrede);          Konsolenausgabe
END;
```

Aufruf

```
CALL kundenanrede_proc(2310);
```

Anm.: - IS und AS können bei der Prozedurdefinition synonym verwendet werden, jedoch nicht bei Cursor, Tabellen und Viewdefinitionen.
- "set serveroutput on" ist zur Anzeige der Konsolenausgabe dbms_output.put_line(...) erforderlich (vgl. Folie 11)

SQLDeveloper – Prozeduren anlegen



Gespeicherte Prozedur- Aufruf

View KundenAus Dortmund

Wohnortwechsel

View KundenAus Bochum

IN → Prozedur Wohnortswechsel → OUT

IN-Parameter | IN OUT-Parameter | OUT-Parameter

```
CALL Wohnortswechsel (8524, 'Dortmund', 'Bochum')
```

Oracle besitzt ein Exception-Mechanismus, daher wird bei der Implementierung kein Rückgabeparameter benötigt.

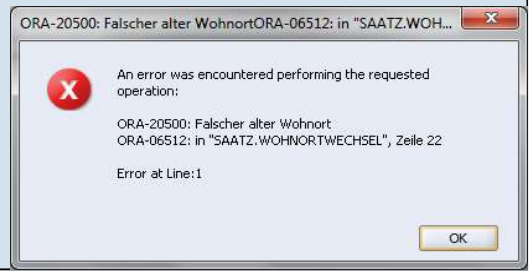
Gespeicherte Prozedur Wohnortwechsel

```
CREATE PROCEDURE Wohnortwechsel (
                Knr         IN     INT,
                alterOrt    IN     VARCHAR2,
                neuerOrt    IN     VARCHAR2)
IS
     wohnort              VARCHAR2(200);
     falscherWohnort      EXCEPTION;
BEGIN
     SELECT  Ort  INTO wohnort
      FROM  Kunde
     WHERE  Kundennummer = Knr;
     IF alterOrt = RTRIM(wohnort,' ') THEN
            UPDATE Kunde SET Ort = neuerOrt
            WHERE  Kundennummer=Knr;
      ELSE
            RAISE falscherWohnort;
      END IF;
EXCEPTION
      WHEN falscherWohnort
      THEN raise_application_error
           (-20500,'Aktueller Wohnort fehlerh.');
END;
```

Selektion des bisherigen Wohnortes

Prüfung der Eingaben
Änderung des Wohnortes

Fehlermeldung werfen



Beispiel: Änderungsoperationen durch Prozeduren

Aufruf der Prozedur Preisänderung aus der Prozedur Sonderangebot:

```
CREATE OR REPLACE PROCEDURE SONDERANGEBOT (anr IN INTEGER, prozent IN INTEGER)
IS sonderpreis NUMBER;
BEGIN
  PREISAENDERUNG(anr,prozent, sonderpreis);
  dbms_output.put_line('Der Sonderpreis des Artikels: ' || anr || ' ist: ' || sonderpreis );
END SONDERANGEBOT;
```

Durchführung der Preisänderung:

```
PROCEDURE PreisAenderung (anr IN INTEGER, aenderung IN Number, neuerPreis OUT Number)
AS
BEGIN
   UPDATE Artikel
   SET preis= preis*(1+aenderung/100)
   WHERE Artikelnummer=anr
   RETURNING preis INTO neuerPreis;
END;
```

Wirkung: neuerPreis:=preis

In dieser Prozedur wird der Preis eines Artikels um einen Prozentsatz geändert.
Der neue (geänderte) Preis wird über einen OUT-Parameter zurückgeliefert.

Zusammenfassung

	Gespeicherte Prozedur	Gespeicherte Funktion
Aufruf	CALL	SELECT
Aufruf-parameter	IN INOUT OUT	IN
Rückgabe-werte	Mehrere OUT-Parameter	Ein Wert
Erlaubte Befehle	DRL DML DDL DCL	DRL
Aufruf von	Funktionen Prozeduren	Funktionen

i.d.R. nicht direkt portierbar zwischen verschiedenen Datenbanken

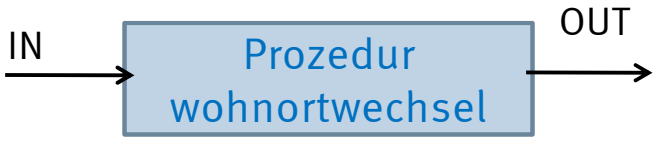
Programmierkonzepte (Datenbank)

- Gespeicherte Funktionen (stored function)

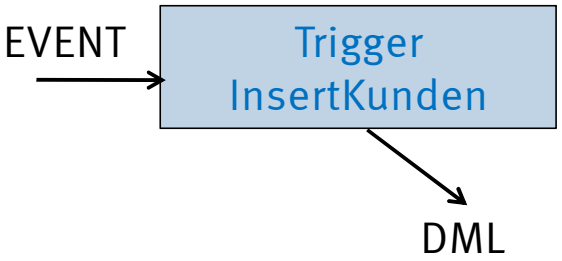
```
SELECT kundenanrede(Kundennummer)
FROM    Kunde
```

- Gespeicherte Prozeduren (stored procedures)

```
CALL kundenliste('Dortmund')
```



- Eventgesteuerte Prozeduren (Trigger)



Vielen Dank
für Ihre Aufmerksamkeit !