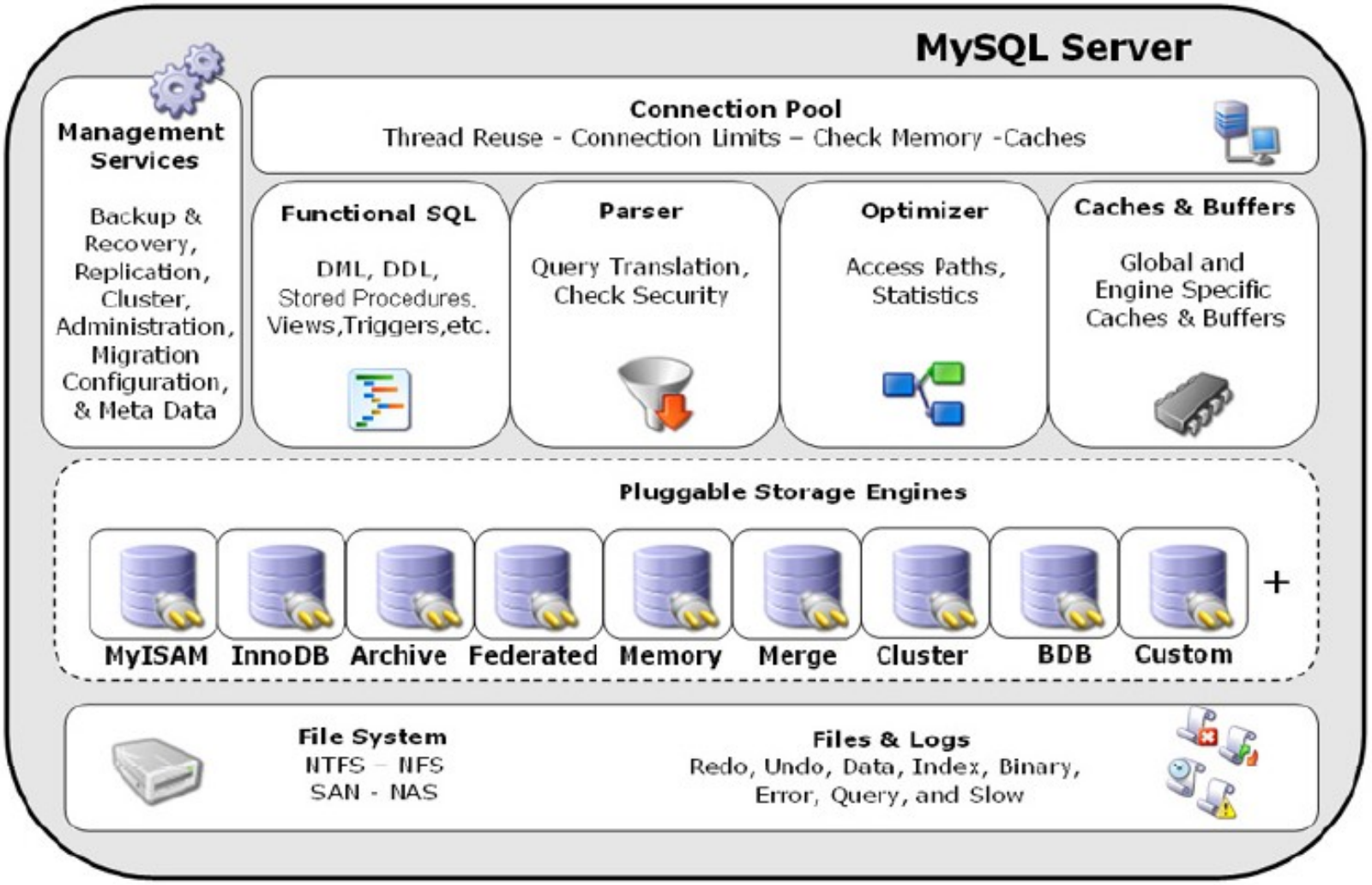
MySQL

View,Stored Procedure, Trigger, Information Schema

[demo/sql-is_uni/sql-kurs2-mysql-trigger-views-procs.sql]

Mysql Overview



Views

- Komplexe SQL-Anweisungen vereinfachen
- Zugriffsschutz
  - View enthält nur bestimmte Spalten einer Tabelle

Views

- CREATE TABLE preis (anz INT, preis INT);
- INSERT INTO preis VALUES(3, 50);
- **CREATE VIEW v_preis**
  AS
  SELECT anz, preis, anz*preis AS summe
  FROM preis;
- mysql> **SELECT * FROM v_preis;**

```
+------+-------+-------+
| anz  |  preis |summe|
+------+-------+-------+
| 3    |  50   | 150   |
+------+-------+-------+
```

Views

```
create VIEW v_notenliste AS
select s.matrnr stud_name, s.name,
       v.vorlnr,v.titel, p.name prof_name, note
from   is_studenten s,is_vorlesungen v,is_professoren p,
       is_pruefen prfg
where
       prfg.matrnr= s.matrnr and
       prfg.vorlnr= v.vorlnr and
       prfg.persnr= p.persnr;


select * from v_notenliste;

...
drop view v_notenliste;
```

Views

```
create or replace view v_prof_hat_viele_studenten
as
select persnr, name, count(*) as anz_studenten
from is_professoren p,
     is_vorlesungen v,
     is_hoeren h
     where p.persnr = v.gelesenvon
     and v.vorlnr = h.vorlnr
group by p.persnr, p.name
having count(*) > 4; -- mehr als 4 Hoerer
```

Stored Procedures

- Code **Reusability** and Change Control
- Fast **Performance**
- Easier **Security** Administration

  access control to be handled at the procedure level rather than at the object level.

Stored Procedures

- For example, MySQL stored procedures offer standardoutput ability for SELECT statements

```
delimiter //

create procedure sp_pruefungen()
    select s.name, v.titel, p.note
    from   is_vorlesungen v, is_pruefen p,
           is_studenten s
    where  p.matrnr=s.matrnr and p.vorlnr=v.vorlnr
 //
delimiter ;

call sp_pruefungen();
```

Stored Procedures

```
CREATE PROCEDURE while_endWhile()
BEGIN
DECLARE v1 INT DEFAULT 5;
      WHILE v1 > 0 DO
       ... SET v1 = v1 - 1;
      END WHILE;
END
```

```
IF search_condition THEN statement_list
      [ELSEIF search_condition THEN statement_list] ...
      [ELSE statement_list]
END IF
```

Cursors

- innerhalb von
  - **Stored Procedures and**
  - **Stored Functions**.

- Sehr schneller Zugriff auf Tabellen,..

Cursors

```
CREATE PROCEDURE sp_prof_mit_vielen_studenten_speichern()
BEGIN
DECLARE done INT DEFAULT 0;
DECLARE vpersnr INT;
DECLARE vname varchar(30);
DECLARE vanz_studenten INT;
DECLARE cur1 CURSOR FOR select persnr, name, anz_studenten
   from v_prof_hat_viele_studenten;
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
   -- NOT FOUND
OPEN cur1;
WHILE done = 0 DO
   FETCH cur1 INTO vpersnr, vname, vanz_studenten;
   IF done = 0 THEN
    INSERT INTO is_uni.is_prof_hat_viele_studenten values
   (vpersnr,vname, vanz_studenten, CURDATE());
   END IF;
END WHILE;
CLOSE cur1;
END
```

Cursors

```
CREATE PROCEDURE curdemo()
BEGIN
DECLARE done INT DEFAULT 0;
DECLARE a CHAR(16);
DECLARE b,c INT;
DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
     //NOT FOUND

OPEN cur1;
OPEN cur2;
SET done= 0;
REPEAT
   FETCH cur1 INTO a, b;
   FETCH cur2 INTO c;
   IF NOT done THEN
    IF b < c THEN
        INSERT INTO test.t3 VALUES (a,b);
    ELSE
        INSERT INTO test.t3 VALUES (a,c);
     END IF;
   END IF;
UNTIL done END REPEAT;
CLOSE cur1; CLOSE cur2;
END
```

SQLSTATE

- SQLWARNING is shorthand for all SQLSTATE codes that begin with 01.
- NOT FOUND is shorthand for all SQLSTATE codes that begin with 02.
- SQLEXCEPTION is shorthand for all SQLSTATE codes not caught by SQLWARNING or NOT FOUND.

Trigger

```
delimiter //

CREATE TRIGGER trg_bi_user_insert
before insert on is_user
for each row
begin
   set NEW.pw_verschluesselt=
  md5(NEW.pw_verschluesselt);
end;
 //

delimiter ;

insert into is_user values ( 'guest','comein');
```

Trigger

- You can refer to columns in the table associated with the trigger by using the aliases **OLD** and **NEW**.
- OLD.*col_name* refers to a column of a an existing row before it is updated or deleted.
- NEW.*col_name* refers to the column of a new row to be inserted or an existing row after it is updated.

Trigger

```
CREATE TRIGGER trg_bu_keinedegradierung
Before update on is_professoren
For each row
Begin
   if OLD.rang=´C4´ then
    SET  NEW.rang=´C4´;
   end if;
   if OLD.rang= ´C3´ and NEW.rang= ´C2´ then
    SET NEW.rang=´C3´;
   end if;
   if NEW.rang is NULL then
    SET NEW.rang= OLD.rang;
    end if;
End
//
-- DER USER, der den Trigger ausführt muss das GLOBALE
   RECHT SUPER haben!
GRANT SUPER ON *.* TO 'is_uni'@'localhost';
```

Information Schema

- Der System-Katalog speichert alle Informationen (Tabellen, Views, …) zur Datenbank (auch DataDictionary genannt)

- ```
  SELECT table_name, table_type, engine
    FROM  information_schema.tables
    WHERE table_schema = 'is_uni'
    ORDER BY table_name DESC;
  ```
  ```

Information Schema

```
SELECT
     T.TRIGGER_NAME, T.EVENT_MANIPULATION, T.TRIGGER_SCHEMA,
      T.EVENT_OBJECT_TABLE, T.ACTION_STATEMENT
FROM
    INFORMATION_SCHEMA.TRIGGERS T
```

Information Schema

```
SELECT
    V.TABLE_SCHEMA,
    V.TABLE_NAME,
    V.VIEW_DEFINITION
FROM
    INFORMATION_SCHEMA.VIEWS V
```

Information Schema

```
SELECT
        R.ROUTINE_SCHEMA,
        R.ROUTINE_NAME,
        R.ROUTINE_TYPE,
        R.ROUTINE_BODY,
        R.ROUTINE_DEFINITION,
        R.SQL_PATH,
        R.EXTERNAL_NAME,
        R.EXTERNAL_LANGUAGE,
        R.CREATED, R.LAST_ALTERED,
        R.DEFINER
FROM
        INFORMATION_SCHEMA.ROUTINES R
```

Übungen

- demo/sql-is_uni/sql-kurs2-mysql-trigger-views-procs.sql

Oracle

Prozedurales SQL (PL/SQL)
Function, Procedure
Package

Prozedurales SQL: PL/SQL

'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
   Vereinbarungen
begin
   Befehle
exception
   Fehler-Behandlung
end;
```

Prozedurales SQL: PL/SQL

Variablendeklaration

Deklaration	Beispiel
`varname type;`	`birthdate DATE;` `name VARCHAR2(20) NOT NULL := 'Huber';`
`constname CONSTANT type := wert;`	`pi CONSTANT REAL := 3.14159;`
Datentypen: * alle SQL-Datentypen * Subtypen	 CHAR, NUMBER, ... z.B. bei NUMBER: DECIMAL, FLOAT, ...
`BOOLEAN`	`switch BOOLEAN NOT NULL := TRUE;`

`%TYPE`	`balance NUMBER(7, 2);`
	`min_bal balance%TYPE;`
	`varname table.attrib%TYPE`

`%ROWTYPE`	`/* employee(name VARCHAR(20),`
	`salary NUMBER(7,2),`
	`dept NUMBER(3)); */`
	
	`emp_rec employee%ROWTYPE;`

Prozedurales SQL: PL/SQL

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(...)`, ...)

Prozedurales SQL: PL/SQL

Ausnahmebehandlung (exception handling)

- Mechanismus zum Abfangen und Behandeln von Fehlersituationen
- Interne (= Oracle) Fehlersituationen
  - Treten auf, wenn das PL/SQL-Programm eine ORACLE-Regel verletzt oder eine 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`, ...

Prozedurales SQL: PL/SQL

- 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;
```

Prozedurales SQL: PL/SQL

- 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');`

Exception: Beispiel

```
declare
  v_bezeichnung    einheit.bezeichnung%TYPE;
  EINHEIT_FEHLER   EXCEPTION;
  PRAGMA EXCEPTION_INIT (EINHEIT_FEHLER, -20100);
begin

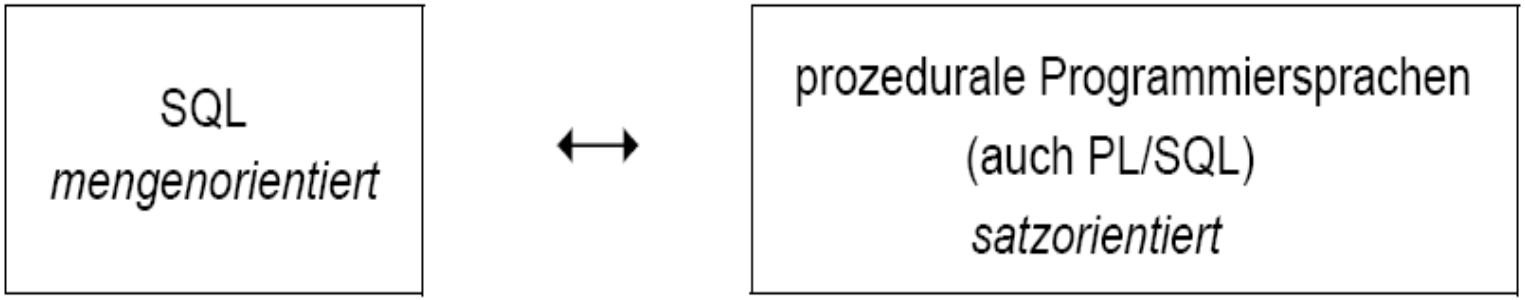
  select bezeichnung into v_bezeichnung
    from einheit
    where einheit_kurz = 'm';

  if v_bezeichnung <> 'Meter' then
    raise_application_error (-20100, 'Bezeichnung falsch');
  end if;

exception when EINHEIT_FEHLER then
  ...
end;
```

Cursor-Konzept:

Problem



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

2 Möglichkeiten

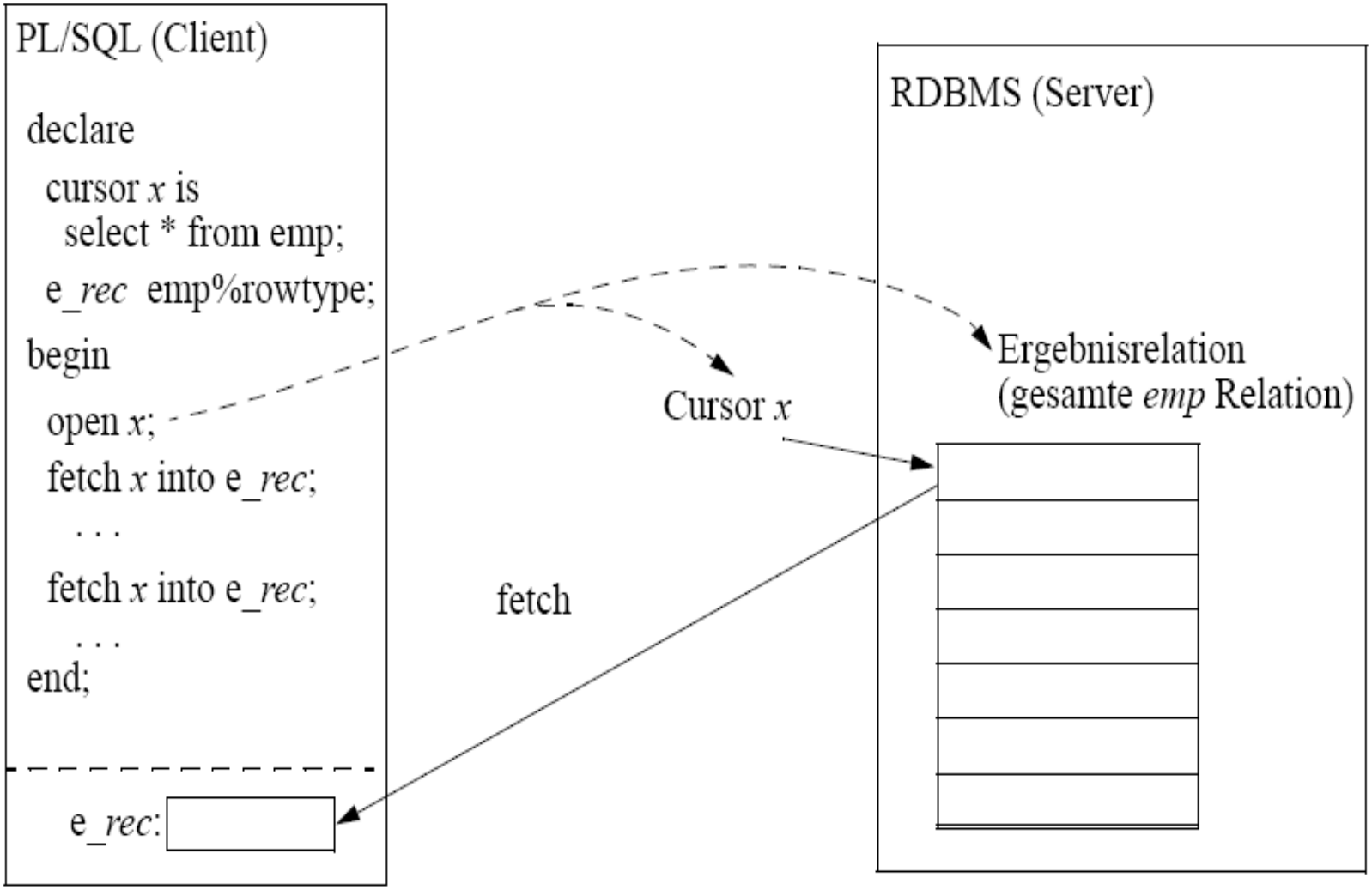
- 1-Tupel-Befehle für Anfragen, die max. 1 Tupel zurückliefern (z.B. `select` ... `into` ...)
- Cursor: Elementweises Durchlaufen der Ergebnismenge und Einlesen jeweils eines Tupels in eine Variable

Cursor-Befehle

```
DECLARE
      CURSOR c1 IS                          -- Deklaration (Verbinden des Cursors
      SELECT n1 from data_table                mit einer Anfrage)
      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;
```



(*e_rec* ist eine Record-Variable, die genau dem Tupel-Typ von *emp* entspricht).

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

PL/SQL: Cursor-FOR-Loops

```
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;
```

Prozedurales SQL: PL/SQL

- 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.
  - 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!

Prozedurales SQL: PL/SQL

- 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.

Prozedurales SQL: PL/SQL

- 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
- Ein Beispiel:

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

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

Packages: PL/SQL

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

Eigenschaften

- Modulkonzept bietet mächtige Strukturierungsmöglichkeiten
- Package ist als Objekt 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

Packages: PL/SQL Beispiel

```
-- SCHNITTSTELLEN BESCHREIBUNG
-- ------------------------------------------------------------------------
create or replace package perf as
  procedure start_timer;
  procedure stop_timer;
  function get_time return number; -- Zeit in Sekunden
end perf;
/
-- IMPLEMENTIERUNG
-- ------------------------------------------------------------------------
create or replace package body perf as
  SEK_JE_TAG constant number := 86400;      -- Var., Cursor-Dekl.
  v_start_time date := NULL;                -- lokal zum body
  v_stop_time date := NULL;

procedure start_timer is                    -- Methode start_timer
begin
  v_start_time := sysdate;
  v_stop_time := NULL;
end;
```

```
procedure stop_timer is                    -- Methode start_timer
begin
  v_stop_time := sysdate;
end;


function get_time return number is         -- Methode get_time
  v_diff_time number;
begin
  if v_start_time is NULL then
    return NULL;
  end if;
  if v_stop_time is NULL then
    v_diff_time := sysdate - v_start_time;
  else
    v_diff_time := v_stop_time - v_start_time;
  end if;
  return v_diff_time * SEK_JE_TAG;
end;

end perf;
```

Erstellen/Aufrufen von Packages

- Befehl **`CREATE PACKAGE`** bewirkt in etwa dieselben Aktionen wie **`CREATE PROCEDURE/FUNCTION`** (siehe S. 70)
- Editieren in Textdatei
- Ausgabe zu Debugging-Zwecken mit vordefiniertem Package **`dbms_output`**
- Aufruf von Package-Prozeduren/Funktionen
  - aus SQL*PLUS:

    ```
    execute <benutzer_name>.<package_name>.<proc_name>(...)
    ```

  - aus PL/SQL:

    ```
    <benutzer_name>.<package_name>.<proc_name>(...);
    ```

proc/func Aufruf mit Toad

-- so ruft man proc/func/pack im SQL-Window von Toad auf
-- alles markeiren und F9
-- es geht ein fenster auf und man gibt einen Wert für die Va. :val ein
-- dies ist ein sehr einfaches Demo
--

```
DECLARE
  RetVal NUMBER;
BEGIN
  perf.START_TIMER();

  RetVal := IS_UNI.inkrement ( :val );
  dbms_output.put_line('StartWert'|| RetVal);

  WHILE RetVal <10000000 LOOP
              RetVal := IS_UNI.inkrement ( RetVal );
  END LOOP;

  perf.STOP_TIMER();

  dbms_output.put_line('EndWert'|| RetVal || ' ' ||
   perf.GET_TIME() || ' Sekunden');
END;
```

Verteilte Datenbanken

Datenbanken werden über sogen. database link Verbindungen zu einer (virtuellen) verteilten Datenbank zusammengeschlossen.

Um die Ortstransparenz zu erreichen, werden

- Synonyme bzw.
- Views verwendet.

Mit einer gewöhnlichen View greift man direkt auf die remote liegenden Daten zu.

Will man lieber eine lokale Kopie, die sich selbständig in regelmäßigen Zeitabständen 'updatet', verwendet man sogen. Materialisierte Views (auch snapshots genannt)

weiteres siehe: vert. DB

Vert. DB: Beispiel

```
drop database link oraxx.sbg;

-- Database link erzeugen
create database link oraxx.sbg
connect to is_uni identified by comein
using 'oraxx.sbg' ;

-- teste den Zugriff auf die entfernte DB
select * from is_studenten@oraxx.sbg

-- ortstransparenz durch synonym
create synonym syn_is_studenten for is_studenten@oraxx.sbg
select * from syn_is_studenten

-- ortstransparenz durch view
create view v_is_studenten as
select *
from is_studenten@oraxx.sbg

select * from v_is_studenten
```

Vert. DB: Beispiel

```
-- materialisierte view
-- 1 Minute: 1440<= (3600*24) / 60
create materialized view mv_is_studenten
REFRESH COMPLETE
START WITH SYSDATE
NEXT sysdate + 1/1440
as
select *
from is_studenten@oraxx.sbg

select * from mv_is_studenten

-- Daten im Original einfügen
insert into is_studenten@oraxx.sbg (matrnr, name, semester)
values (12345, 'hofmann',10);
commit;


select * from v_is_studenten where name='hofmann'

select * from mv_is_studenten where name='hofmann'
```

Vert. DB: ein Join über verteilte Daten

```
create VIEW v_notenliste AS
select s.matrnr stud_name, s.name,
       v.vorlnr,v.titel, p.name prof_name, note
from
       is_studenten@ora00.sbg s,
       is_vorlesungen@ora00.sbg v,
       is_professoren@ora01.sbg p,
       is_pruefen@ora01.sbg prfg
where
       prfg.matrnr= s.matrnr and
       prfg.vorlnr= v.vorlnr and
       prfg.persnr= p.persnr;


select * from v_notenliste;
```