

## Stored Procedures

- ❑ With the command **create**, functions and procedures can be stored in the DBMS in a translated form and called on request.
- ❑ advantage: no anew translation of the query necessary
- ❑ Declaration is done according to the aforementioned pattern.
- ❑ cursor variables
  - Frequently, it is favorable to transmit the results of a stored procedure through cursor variables to the calling PL/SQL program.
- ❑ A cursor is a reference to a list of data records.
- ❑ two types of cursor variables
  - strong type
    - type personCursorType is ref cursor person%rowtype;**
  - weak type
    - type allCursorType is ref cursor;**
- ❑ variable declaration as usual
- ❑ At the time of its declaration the cursor variable does not have a relationship to a query.

## ❑ Bindung of a cursor variable to queries

When opening a cursor, the variable is bound to a query.

```
open personCursor for select * from persons where salary > 2000
```

## ❑ usual application

- opening of a cursor variable in the stored procedure/function
- handing over of the cursor to the AP, which processes the records

## ❑ Instead of the many advantages of cursor variables, there are still many limitations:

- A cursor variable may not be opened in the update mode.
- Type **ref cursor** is only known in PL/SQL but not in SQL.

## ❑ stored functions in SQL

- Stored functions can be declared and called in SQL with the following limitations:
  - + The functions do not contain grouping operations.
  - + All data types of the input and of the output must be known in the DBMS.
- example for the declaration of a stored function

```
create function simple (x in int) return int as begin return x/101; end simple;
```

- example of an SQL query using this function

```
select Name, simple(salary) from Persons;
```

- example: program segment which yields the information about the employee with the highest salary

**declare**

```
lname      employee.lastname%type;  
fname      employee.firstname%type;  
addr       employee.address%type;  
esalary    employee.salary%type;
```

**begin**

```
select lastname, firstname, address, salary  
into lname, fname, addr, esalary  
from employee  
where salary = (select max(salary) from employee);  
dbms_output.put_line(lname, fname, addr, esalary);
```

**exception**

```
when others then  
dbms_output.put_line("Error detected!");
```

```
end;
```

- example: program segment which increases the salary of employees, whose salary is below the average salary, by 10% and which outputs the average salary, if it exceeds 30000 Dollar after the previous update.

**declare**

```
avg-salary number;
```

**begin**

```
select avg(salary) into avg-salary from employee;
```

```
update employee
```

```
  set salary = salary * 1.1
```

```
  where salary < avg-salary;
```

```
select avg(salary) into avg-salary from employee;
```

```
if avg-salary > 30000 then
```

```
  dbms_output.put_line("Average salary is " || avg-salary);
```

```
end if;
```

```
commit;
```

**exception**

```
  when others then dbms_output.put_line("Update error!"); rollback;
```

```
end;
```

- example: Calculate the salary increases depending on the current salaries of employees.

**declare**

```
cursor EmpCursor is select salary from employee for update of salary;  
EmpSal      employee.salary%type;
```

**begin**

```
open EmpCursor;
```

```
fetch EmpCursor into EmpSal;
```

```
while not EmpCursor%notfound
```

```
    if EmpSal > 60000 then      update employee set salary = salary * 1.1  
                                where current of EmpCursor;
```

```
    elsif EmpSal > 50000 then  update employee set salary = salary * 1.15  
                                where current of EmpCursor;
```

```
    else update employee set salary = salary * 1.20  
        where current of EmpCursor;
```

```
    end if;
```

```
    fetch EmpCursor into EmpSal;
```

```
end loop;
```

```
end;
```

## 6.5 Pascal/R

- ❑ PASCAL/R is the attempt to integrate concepts of relational query languages into the programming language PASCAL in a most possible homogeneous way and with minimum extensions.

### Schema description

- ❑ Tuples correspond to records with scalar data types
- ❑ Relation types can be defined using record types by fixing key attributes.
- ❑ example:

```
type EmplRecType = record
    EmplNo, EmplStatus : Integer;
    EmplName : String;
end;
    EmplRelType = relation EmplNo of EmplRecType;
var Employees : EmplRecType;
```

- ❑ New attribute domains can be explicitly modeled by introducing corresponding types.

## Elementary operations on relations

### ❑ insertion operator

`rel1 :+ rel2;`

*rel1* and *rel2* must be relation variables of the same type. All tuples of *rel2* whose key values do not occur in a tuple of *rel1* are inserted into *rel1*.

### ❑ deletion operator

`rel1 :- rel2;`

*rel1* and *rel2* must be relation variables of the same type. All Tuple of *rel1* which are also contained in *rel2* are removed from *rel1*.

### ❑ replacement operator

`rel1 :& rel2;`

*rel1* and *rel2* must be relation variables of the same type. Each tuple of *rel1* whose key attributes are contained in a tuple of *rel2* is replaced by this tuple.

### ❑ assignment operator

`rel1 := rel2;`

❑ basic relation constructor

Let *rec* be a record variable of type *RecType*. Then {*rec*} is a relation (relation constant, relation value) containing just this tuple. Its type is compatible with all relation types declared over *RecType*.

❑ Let *rel* be a variable of such a relation type. Then, for example,

$$\text{rel} := \{\text{rec}\} \quad \text{or} \quad \text{rel} := \{+\text{rec}\}$$

are correct statements.

❑ {} denotes the empty relation.

❑ If a tuple consists only of constants, it is, for example, allowed to write:

$$\text{rel} := \{<4320, 2, \text{"Smith, Ben"}>\}$$

❑ iteration over relations

– **<foreach-statement> ::=** **foreach** <control-record-variable>  
**in** <range-relation-variable>  
**do** <statement>

– <control-record-variable> is implicitly declared according to the relation variable. In the loop, each tuple of the relation is assigned exactly once to the record variable (in “random” order). In the statement part neither the relation in total nor the key attributes of the record variable may be changed.

- query example: Find all employees with status 2
- ```

type EmplRelType = (as above)
var Employees, Result : EmplRecType;
begin
    ...
    Result := {};
    foreach Empl in Employees do
        if Empl.EmplStatus = 2 then Result :=+ {Empl}
    end;

```

## Predicates over relations

### □ structure

```

<predicate> ::= <quantifier> <control-record-variable>
               in <range-relation-variable> (<log. expression>)

<quantifier> ::= some | all

<log. expression> ::= <term> | <term> <log. operator> <log. expression>

<log. operator> ::= and | or

```

- ❑ Terms consist of components of the control-record-variables, program variables and constants. A predicate is also a term.
  
- ❑ All tuple variables are restricted to (ranges of) relations. The operator **not** is not allowed (like in Condition Boxes in QBE). This ensures safe expressions.

❑ example:

```
type TimeBlock =          (0810, 1012, 1214, 1416,1618);
      WeekDay =           (Monday, ..., Friday);
      TimeTableRecType = record
                          EmplNo, LectId : Integer;
                          Day : WeekDay;
                          Time : TimeBlock;
                          Room : String;
                          end;
      TimeTableRelType = relation LectId, Day of TimeTableRecType;
var TimeTable : TimeTableRelType;
(other declarations like before)
```

- ❑ query example: Find all professors (EmplStatus = 3) holding lectures on Fridays.

**begin**

...

Result := {};

**foreach** Empl **in** Employees **do**

**if** Empl.EmplStatus = 3 **and some** Date **in** TimeTable

        ((Empl.EmplNo = Date.EmplNo) **and** (Date.Day = Friday))

**then** Result :=+ {Empl}

**end**

- ❑ general relation constructor

<gen. rel. constructor> ::= { **each** <control-record-variable>  
                                  **in** <range-relation-variable> : <log. expression> }

- yields a relation of the same type as range relation variable
- contains all tuples fulfilling the logical expression, can also contain predicates

- ❑ query example: formulation of the previous query with substitution of the foreach-loop

```
Result := { each Empl in Employees :
            Empl. EmplStatus = 3 and some Date in TimeTable
            ((Empl.EmplNo = Date.EmplNo) and (Date.Day = Friday)) }
```

- ❑ Further generalization
  - several relations (cartesian product, join)
  - selection of tuple components (projection)
  - Type of the result relation must be declared correspondingly to be able to assign the result to a variable.
- ❑ query example: Determine the names of professors, the lecture id, and the time for lectures on Fridays.

```
var Result : relation LectId of record
                Name : String;
                LectId : Integer;
                Time : TimeBlock
                end;

begin
    Result := {(Empl.EmplName, Date.LectId, Date.Time) of
                each Empl in Employee, each Date in TimeTable :
                Empl.EmplNo = Date.EmplNo and Date.Day = Friday};

end
```

## Coupling program – database

- ❑ For the program the database is an external variable:

```
program DBProgramm(UniDB);
```

```
type ...
```

```
    EmplRelType = ...
```

```
    TimeTableRelType = ...
```

```
var UniDB : database Employees : EmplRelType;
```

```
                TimeTable : TimeTableRelType end;
```

```
    Result : EmplRelType;
```

```
begin with UniDB do ...
```

```
end.
```

- ❑ advantageous
  - embedding of relational concepts in Pascal/R rather elegant and homogeneous
- ❑ disadvantageous
  - language cannot be used interactively
  - special compiler and special runtime system necessary

## 6.6 Further Coupling Alternatives

### Skript languages

- ❑ popular approach for the construction of APs (e.g. Visual Basic)
- ❑ disadvantageous
  - no strict typing
  - no standards (bad maintenance of application software)
  - mixing of different concepts from fields like databases, imperative programming, user interfaces and rules
- ❑ advantageous
  - fast development of APs with graphical user interfaces
  - comfortable development environments

# 7. Data Integrity

## 7.1 Introduction

### Integrity constraints

- ❑ **Integrity constraints** (ICs)
  - are an instrument to ensure that changes of the database by authorized users do not lead to a loss of data consistency.
  - serve for a restriction of the database states to those ones that really exist in the real world.
  - are semantically derivable from the posed data model and can therefore already be specified during the creation of the schema.
- ❑ advantages
  - Consistency conditions are specified only once.
  - Consistency conditions are checked automatically by the DBMS.
  - APs do not need to care about a check of the conditions.
- ❑ **Static integrity constraints** relate to restrictions of the possible database states, **dynamic integrity constraints** to restrictions of the possible database state transitions.

- example for static IC: German professors may only have ranks C2, C3 or C4.
- example for dynamic IC: Professoren may only be promoted, but not demoted. Their rank may not be set from C4 to C3, for example.

## Examples for ICs

- No customer name may appear more than once in the relation “customers”.
- Each customer name in the relation “orders” must appear in the relation “customer”.
- No account of a customer is allowed to be less than USD -100.00.
- The account of Mr White may not be overchecked.
- Only those products can be ordered for which at least one supplier exists.
- The bread price may not be increased.