RAWDATA
Section 1

SQL part 3
Programming & Advanced features

Henrik Bulskov & Troels Andreasen

Programming & Advanced features

- ❑ Accessing SQL From a Programming Language
  - – JDBC, ODBC and ADO.NET
    - ADO.NET with C# will be covered in more detail in section two
- ❑ Programming the database
  - – Functions and Procedural Constructs in SQL
  - – Triggers in SQL

- ❑ Recursive Queries
- ❑ Advanced aggregation (Ranking, Windowing)
- ❑ Data Analysis and OLAP

JDBC and ODBC and ADO.NET

- ❑ API (application-program interface) for a program to interact with a database server
- ❑ Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables

- ❑ JDBC (Java Database Connectivity)
  - works with Java
- ❑ ODBC (Open Database Connectivity)
  - works with C, C++, C#, and Visual Basic
- ❑ ADO.NET
  - works with the .NET framework
  - will be used with C# on RAWDATA
    - especially with what's called Entity-Framework and LINQ
  -

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- Model for communicating with the database:
  1) Open a connection
  2) Create a “statement” object
  3) Execute queries using the Statement object to send queries and fetch results
  4) Extract data from result set
  5) Close connection
  – (Use exception mechanism to handle errors)

Java & JDBC Code

```
//STEP 1. Import required packages
import java.sql.*;

public class FirstExample {
  static final String DB_URL = "jdbc:postgresql://localhost:5432/university"; // a JDBC url

  //static final String DB_URL
  static final String USER = "postgres";
  static final String PASS = "toor";

  public static void main(String[] args) {
  Connection conn = null;
  Statement stmt = null;
  try{

    //STEP 1: Register JDBC driver
    Class.forName("org.postgresql.Driver");

    //STEP 2: Open a connection
    System.out.println("Connecting to database...");
    conn = DriverManager.getConnection(DB_URL,USER,PASS);

    //STEP 3: Execute a query
    System.out.println("Creating statement...");
    stmt = conn.createStatement();
    String sql;
    sql = "SELECT id, name, salary FROM instructor";
    ResultSet rs = stmt.executeQuery(sql);

    //STEP 4: Extract data from result set
    while(rs.next()){
      //Retrieve by column name and display values
      System.out.println("ID: " + rs.getString("id") + " " + rs.getString("name") + " " + rs.getInt("salary"));
    }
```

```
  //STEP 5: Clean-up environment
      rs.close();
      stmt.close();
      conn.close();
    }catch(Exception e){
      //Handle errors
      e.printStackTrace();
    }
    System.out.println("Goodbye!");
  }//end main
}//end FirstExample
```

```
Connecting to database...
Creating statement...
ID: 10101 Srinivasan 65000
ID: 12121 Wu 90000
ID: 15151 Mozart 40000
ID: 22222 Einstein 95000
ID: 32343 El Said 60000
ID: 33456 Gold 87000
ID: 45565 Katz 75000
ID: 58583 Califieri 62000
ID: 76543 Singh 80000
ID: 76766 Crick 72000
ID: 83821 Brandt 92000
ID: 98345 Kim 80000
Goodbye!
```

ADO.NET

- ❑ The ADO.NET API provides functions to access data similar to the JDBC functions.
- ❑ Thus ADO.NET allows access to results of SQL queries
- ❑ A similar model for communicating with the database:

  1) Open a connection
  2) Create a “statement” object
  3) Execute queries using the Statement object to send queries and fetch results
  4) Extract data from result set
  5) Close connection

C# & ADO.NET

```
ID: 10101 Srinivasan 65000
ID: 12121 Wu 90000
ID: 15151 Mozart 40000
ID: 22222 Einstein 95000
ID: 32343 El Said 60000
ID: 33456 Gold 87000
ID: 45565 Katz 75000
ID: 58583 Califieri 62000
ID: 76543 Singh 80000
ID: 76766 Crick 72000
ID: 83821 Brandt 92000
ID: 98345 Kim 80000

Press any key to continue...
```

```
using System;
using Npgsql;

namespace AdoExample
{
    class Program
    {
        static void Main(string[] args)
        {
            var connString = "Host=localhost;Username=troels;Password=troels;Database=uni";

            using (var conn = new NpgsqlConnection(connString))
            {
                conn.Open();
                // Retrieve all instructors
                using (var cmd = new NpgsqlCommand("SELECT id, name, salary FROM instructor", conn))
                using (var rdr = cmd.ExecuteReader())
                    while (rdr.Read()){
                        Console.Write("ID: {0} {1} {2} \n", rdr[0], rdr[1], rdr[2]);
                    }
            }
        }
    }
}
```

Functions and Procedural Constructs in SQL

Procedural Extensions and Stored Procedures

- ❑ SQL provides a **module** language
  - – Permits definition of functions and procedures in SQL
- ❑ Functions
  - – write your own functions and add them to the database
  - – use them like any function predefined by the DBMS, that is, within expressions
- ❑ Stored Procedures
  - – store procedures in the database
  - – execute them by ”calling” them from applications or interfaces to the DBMS
  - – this permits external applications to operate on the database without knowing about internal details
  - – you can, for instance, make your own dedicated API that provides functionality but hides the database structure
- ❑ Triggers
  - – you can add special procedures that are executed automatically by the system as a side effect of a modification to the database

Procedural Extensions and Stored Procedures

- ❑ PostgreSQL specialities
  - PostgreSQL provides probably the most advanced framework and language extension for adding functions and procedures to the DBMS
  - PostgreSQL does not include an explicit notion of Stored Procedure, but (as in other languages) you can consider a function that does not return anything to be a Procedure
    - Stored Procedure ~ Function of type void
  - most DBMS provide a **call** statement to execute a stored procedure,
  - PostgreSQL allow Stored Procedures (void functions) and functions to be invoked by SELECT-expressions or by using a special **perform** command)

Functions and Procedures

- ❑ Since SQL:1999 the standard supports functions and procedures
  - – Functions/procedures can be written in SQL itself, or in an external programming language.
  - – Some database systems (including PostgreSQL) support a particularly useful construct:
    - **table-valued function**, (returning a relation as a result).

- ❑ SQL:1999 also supports a rich set of imperative constructs, including
  - – Loops, if-then-else, assignment, and others

- ❑ Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.

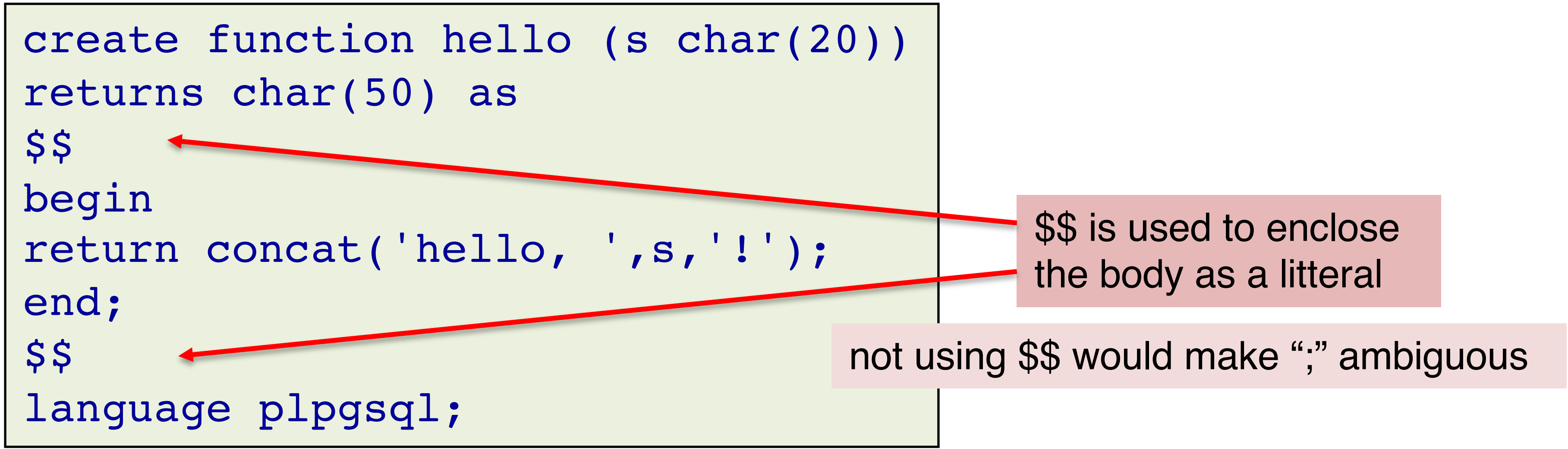
SQL Functions

- ❑ Define a function.

  create function hello **(s char(20))**
  returns char(50)
  begin
  return concat('hello, ',s,'!'**);**
  end;

- ❑ Use the function.

  select hello('world'**)** 'Message to all'**;**



```
create function hello (s char(20))
returns char(50) as
$$
begin
return concat('hello, ',s,'!');
end;
$$
language plpgsql;
```

$$ is used to enclose the body as a litteral

not using $$ would make “;” ambiguous

```
uni=# select hello('world') as "Message to all";
 Message to all
----------------
 hello, world!
(1 row)
```

SQL Functions

```
create function hello (s char(20))
returns char(50) as
$$
begin
return concat('hello, ',s,'!');
end;
$$
language plpgsql;
```

Same function but now used on a table

```
uni=# select hello(name) "Message to all" from instructor;
   Message to all
--------------------
 hello, Srinivasan!
 hello, Wu!
 hello, Mozart!
 hello, Einstein!
 hello, El Said!
 hello, Gold!
 hello, Katz!
 hello, Califieri!
 hello, Singh!
```

SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
 returns integer
 begin
     declare d_count integer;
     select count ( * ) into d_count
     from instructor
     where instructor.dept_name = dept_name
     return d_count;
 end
```

DSC Figure 5.5

- Find the department name and budget of all departments with more that 1 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 1
```

SQL Functions

- Same function, but now using the PL/pgSQL language
- Again count of the number of instructors in that department.

```
create function dept_count (d_name char(20))
returns integer as $$
declare d_count integer;
begin
       select count(*) into d_count
       from instructor
       where instructor.dept_name = d_name;
       return d_count;
end;
$$
language plpgsql;
```

```
uni=# select dept_count('Physics');
 dept_count
------------
          2
(1 row)
```

- or use the dept_count()-function in a where-condition.

```
create function dept_count (d_name char(20))
returns integer as $$
declare d_count integer;
begin
      select count(*) into d_count
      from instructor
      where instructor.dept_name = d_name;
      return d_count;
end;
$$
language plpgsql;
```

```
uni=# select dept_name, budget from department
uni-# where dept_count(dept_name ) > 1;
 dept_name  |  budget
------------+-----------
 Comp. Sci. | 100000.00
 Finance    | 120000.00
 History    |  50000.00
 Physics    |  70000.00
(4 rows)
```

- ❑ or use the dept_count()-function in the select clause

```
create function dept_count (d_name char(20))
returns integer as $$
declare d_count integer;
begin
      select count(*) into d_count
      from instructor
      where instructor.dept_name = d_name;
      return d_count;
end;
$$
language plpgsql;
```

```
uni=# select distinct dept_name, dept_count(dept_name)
uni-# from department;
 dept_name  | dept_count
------------+------------
 Biology    |          1
 Comp. Sci. |          3
 Elec. Eng. |          1
 Finance    |          2
 History    |          2
 Music      |          1
```

SQL Procedures

- The *dept_count* function could instead be written as procedure:

```
create procedure dept_count_proc(in dept_name varchar(20),
                                 out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_name;
end
```

DSC page 175

- The SQL standard suggests that
  - Procedures can be invoked either from an SQL procedure or
  - from embedded SQL, using the **call** statement.
- Postgres
  - does not include a Procedure construct
  - does not include a call statement, but do have a perform
  - only functions with return types can be defined
  - however return type void would correspond to a procedure

SQL Procedures

```
create procedure dept_count_proc(in dept_name varchar(20),
                                 out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_name;
end
```

DSC page 175

- Since this procedure includes a value in the out-parameter *d_count* Postgres requires the definition to be an integer- rather than an void-function

```
create function dept_count_proc (in d_name varchar(20),
                                 out d_count int)
returns integer as
$$
begin
   select count(*) into d_count
   from instructor
   where instructor.dept_name = d_name;
end;
$$
language plpgsql;
```

SQL Procedures

calling the procedure from the command line

```
uni=# select dept_count_proc ('Physics');
 dept_count_proc
-----------------
               2
(1 row)
```

```
create function dept_count_proc (in d_name varchar(20),
                                 out d_count int)
returns integer as
$$
begin
   select count(*) into d_count
   from instructor
   where instructor.dept_name = d_name;
end;
$$
language plpgsql;
```

- Nothing is gained from defining the out-parameter here
  A more straight forward version would be

```
create function dept_count_proc (d_name varchar(20))
returns integer as
$$
begin
   return (select count(*)
   from instructor
   where instructor.dept_name = d_name);
end;
$$
language plpgsql;
```

```
create function dept_count_proc (in d_name varchar(20),
                                 out d_count int)
returns integer as
$$
begin
   select count(*) into d_count
   from instructor
   where instructor.dept_name = d_name;
end;
$$
language plpgsql;
```

SQL Procedures

```
create function dept_count_proc (d_name varchar(20))
returns integer as
$$
begin
   return (select count(*)
   from instructor
   where instructor.dept_name = d_name);
end;
$$
language plpgsql;
```

- or an even simpler as a plain SQL language function

```
create function dept_count_proc (d_name varchar(20))
returns bigint as
$$
   select count(*)
   from instructor
   where instructor.dept_name = d_name;
$$
language sql;
```

Table Functions

- SQL:2003 added functions that return a relation as a result

- Example: Return all instructors of a given department

```
 create function instructors_of (dept_name char(20))
  returns table (  ID varchar(5),
               name varchar(20),
              dept_name varchar(20),
               salary numeric(8,2))
return table
  (select ID, name, dept_name, salary
   from instructor
   where instructor.dept_name = instructors_of.dept_name)
```

DSC Figure 5.6

- Usage

```
select *
from table (instructors_of ('Physics'))
```

Table Functions

- ❑ Same function, but now using the SQL language (in the body)
- ❑ Again used to retrieve instructors in the Physics department.

```
create function instructors_of (dept_name char(20))
             returns table (ID varchar(5),
                    name varchar(20),
                    dept_name varchar(20),
                    salary numeric(8,2)) as
$$
       (select ID, name, dept_name, salary
       from instructor
       where instructor.dept_name = instructors_of.dept_name);
$$
language sql;
```

```
uni=# select * from instructors_of ('Physics');
  id   |   name    | dept_name |  salary
-------+-----------+-----------+----------
 22222 | Einstein  | Physics   | 95000.00
 33456 | Gold      | Physics   | 87000.00
(2 rows)
```

Procedural Constructs

- Conditional statements (**if-then-else**)

- Compound statement: **begin … end**,
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statement

- Loops (among others): **While** and **repeat** statements :

```
declare n integer default 0;
while n < 10 do
        set n = n + 1;
end while;

repeat
        set n = n – 1;
until n = 0
end repeat;
```

- Warning: most database systems implement their own variant of a modular (procedural) language – only inspired by the standard syntax

SQL Procedure, example with WHILE loop

```
drop table if exists foo;
create table foo
(
  id serial primary key,
  val integer
);
```

a table, foo, for testing

defining the procedure

```
create or replace function load_foo()
returns void as
$$
declare
   i_max integer := 4;
   i integer := 0;
   n integer;
begin
  while i < i_max loop
    n:=(random() * 10000);
    insert into foo (val) values (n);
    i:=i+1;
  end loop;
end
$$
language plpgsql;
```

calling and showing the effect

```
uni=# select load_foo();
 load_foo
----------
(1 row)

uni=# select * from foo;
 id | val
----+------
  1 | 3822
  2 | 5438
  3 | 8353
  4 | 9690
(4 rows)
```

SQL Procedure, example (cont.)

- ❑ Notice SQL-details
  - drop … if exists … (very useful in a script you want to run repeatedly)
    - `drop table if exists foo;`
  - auto incrementing primary key
    - `id serial primary key,`
  - declaration and initialization of variable
    - `i_max integer := 4;`
  - while loop to do several DML-statements
    - `while i < i_max loop`
  - random() between 0 and 1 to generate number between 0 and 9999
    - `n:=(random() * 10000);`

Calling a procedure from another

```
drop table if exists foo;
create table foo
(
  id serial primary key,
  val integer
);
```

a table, foo, for testing

defining the procedure that calls the procedure

```
create or replace function test()
returns void as
$$
begin
  perform load_foo();
  perform load_foo();
end
$$
language plpgsql;
```

calling and showing the effect

```
uni=# select test();
 test
------

(1 row)

uni=# select * from foo;
 id | val
----+------
  1 | 8537
  2 | 3004
  3 | 6846
  4 | 3274
  5 | 6931
  6 | 4464
  7 | 8140
  8 | 5080
(8 rows)
```

SQL Procedure, simplified using a FOR loop

```
create or replace function load_foo()
returns void as
$$
declare
   i_max integer := 4;
   i integer := 0;
   n integer;
begin
  while i < i_max loop
    n:=(random() * 10000);
    insert into foo (val) values (n);
    i:=i+1;
  end loop;
end
$$
language plpgsql;
```

```
create or replace function load_foo()
returns void as
$$
begin
  for i in 1..4 loop
    insert into foo (val) values (random() * 10000);
    i:=i+1;
  end loop;
end
$$
language plpgsql;
```

Cursor

- ❑ **cursor**
  - is a control structure that enables traversal of rows in a table
  - a cursor is declared by a query and the table to be traversed is the result of this query
- ❑ **declare**
  - Before a cursor can be used it must be declared (defined).
  - `declare cur1 cursor for select name,salary from instructor;`
- ❑ **open** – perform the query
  - The cursor must be opened for use. This process actually retrieves the data using the previously defined SELECT statement.
  - `open cur1;`
- ❑ **fetch** – get the next row from the table
  - Individual rows can be fetched (retrieved) as needed.
  - `fetch cur1 into a, b;`
- ❑ **close** – close the cursor (clean up)
  - When done, the cursor must be closed.
  - `close cur1;`

SQL Procedure using cursor, example

a table, **vip**, for testing

```
drop table if exists vip;
create table vip  as
  select name, salary
  from instructor;
truncate vip;
```

```
create or replace function curdemo()
returns void as
$$
DECLARE
  rec record;
  cur1 cursor for select name,salary from instructor;
begin
  open cur1;
  loop
    fetch cur1 into rec;
    exit when not found;
    if rec.salary > 81000 then
      insert into vip
          values (rec.name,rec.salary);
    end if;
  end loop;
  close cur1;
end;
$$
language plpgsql;
```

calling and showing the effect

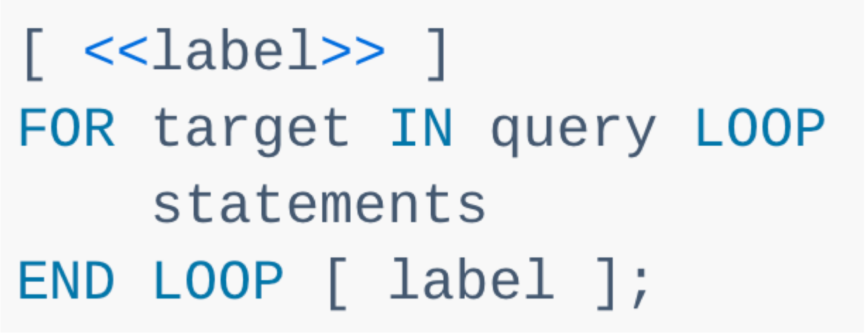
```
uni=# select curdemo();
 curdemo
---------


(1 row)

uni=# select * from vip;
   name    |  salary
----------+----------
 Wu        | 90000.00
 Einstein  | 95000.00
 Gold      | 87000.00
 Brandt    | 92000.00
(4 rows)
```

SQL Procedure using cursor, example(cont.)

- ❑ Notice SQL-details
  - the 4 “using cursors”-issues to remember:
    - **declare, open, fetch, close**
  - conditional statement (fairly standard)
    - `if … then … end if;`
  - a very useful data type **record**:
    - `rec record;`
    - …
    - `fetch cur1 into rec;`
  - another loop construction
    - `loop … exit when … end loop;`

Yet another loop …

- ❑ A very useful loop in PostgreSQL is the following



```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

- ❑ DO-block
  - – anonymous function
  - – very useful for adhoc tasks and for testing expressions
- ❑ raise notice …
  - – very useful for testing expressions

testing with a DO block

```
uni=# do $$
uni$# declare
uni$#     rec record;
uni$# begin
uni$#     for rec in select name
uni$#         from instructor
uni$#     loop
uni$#  raise notice '%', rec.name;
uni$#     end loop;
uni$# end;
uni$# $$;
NOTICE:  Srinivasan
NOTICE:  Wu
NOTICE:  Mozart
NOTICE:  Einstein
NOTICE:  El Said
NOTICE:  Gold
NOTICE:  Katz
NOTICE:  Califieri
NOTICE:  Singh
```

SQL using cursor (now implicit), example

a table, **vip**, for testing

```
-- vip(name, salary)
```

```
create or replace function curdemo2()
returns void as
$$
declare
  rec record;
begin
  for rec in select name,salary from instructor
  loop
    if rec.salary > 81000 then
      insert into vip
        values (rec.name,rec.salary);
    end if;
  end loop;
end;
$$
language plpgsql;
```

calling and showing the effect

```
uni=# truncate vip;
TRUNCATE TABLE
uni=# select curdemo2();
 curdemo2
----------


(1 row)

uni=# select * from vip;
   name    |  salary
----------+----------
 Wu        | 90000.00
 Einstein  | 95000.00
 Gold      | 87000.00
 Brandt    | 92000.00
(4 rows)
```

SQL using cursor (now implicit), ex. (cont.)

- Compare to the cursor example above
  - the loop is changed to
    - `for rec in select name,salary from instructor`
    - `loop`
    - …
    - `end loop`
  - the cursor is replaced by the expression given as argument in the for loop
    - `for rec in select name,salary from instructor`
  - conceptually this is an implicit cursor

External Language Functions/Procedures

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

```
create procedure dept_count_proc(in dept_name varchar(20),
                                 out count integer)
language C
external name ’ /usr/avi/bin/dept_count_proc’

create function dept_count(dept_name varchar(20))
returns integer
language C
external name ‘/usr/avi/bin/dept_count’
```

External Language Functions/Procedures

- ❑ Notice the PostgreSQL **CREATE FUNCTION** statement:

```
CREATE FUNCTION function_name(…)
RETURNS type AS
BEGIN
-- logic
END;
LANGUAGE language_name;
```

- ❑ By default, PostgreSQL supports three languages:
  - – SQL, PL/pgSQL, and C.
- ❑ You can also load other procedural languages
  - – e.g., Perl, Python, and TCL

External Language Routines (Cont.)

- ❑ Benefits of external language functions/procedures:
  - – more efficient for many operations, and more expressive power.
- ❑ Drawbacks
  - – Code to implement function may need to be loaded into database system and executed in the database system's address space.
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data

Why use Stored functions and procedures?

- ❑ Stored functions and procedures (routines) can be particularly useful
  - When multiple client applications are written in different languages or work on different platforms, but need to perform the same database operations.
  - When security is paramount. Banks, for example, use stored procedures and functions for all common operations
  - In addition, you can store libraries of functions and procedures in the database server

  - Provide improved performance. Less information needs to be sent between the server and the client.
  - Tradeoff: increase the load on the database server

Triggers

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.

- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.

Trigger Example – Referential constraint

- E.g. *time_slot_id* is not a primary key of *timeslot,* so we cannot create a foreign key constraint from *section* to *timeslot.*
- Alternative: use triggers on *section* and *timeslot* to enforce integrity constraints
- Figure 5.8 in DSC book:

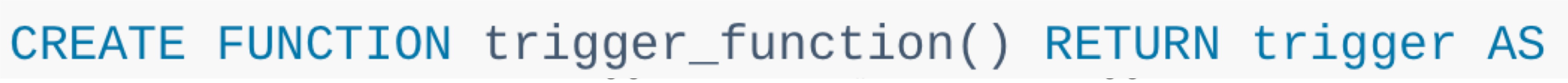
```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
            select time_slot_id
            from time_slot)) /* time_slot_id not present in time_slot */
begin
   rollback
end;
```

Will not work in MySQL
Roolback is not allowed

PostgreSQL Triggers

- ❑ To create a new trigger in PostgreSQL:
  - – Create a trigger function using CREATE FUNCTION statement.
  - – Bind this trigger function to a table using CREATE TRIGGER statement.

- ❑ Create the trigger function

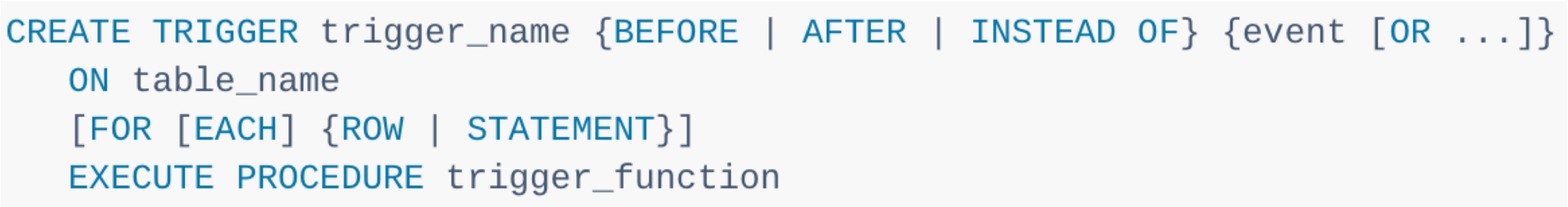


```
CREATE FUNCTION trigger_function() RETURN trigger AS
```

  - – a function similar to an ordinary function,
  - – does not take any arguments
  - – has return **return type trigger**
  - – important variables: **OLD** and **NEW** represent the states of row in the table before or after the triggering event.

PostgreSQL Triggers

- ❑ Create the trigger
  - – use the CREATE TRIGGER statement:

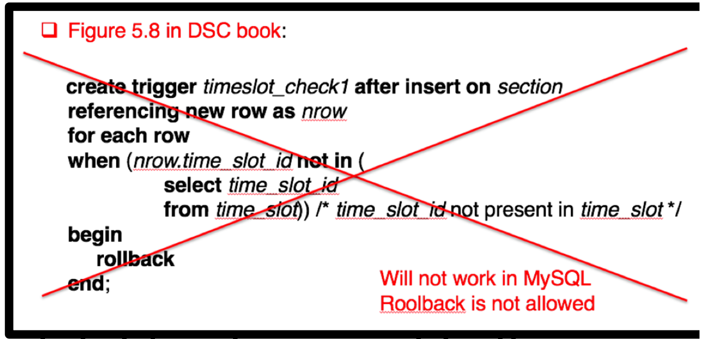


```
CREATE TRIGGER trigger_name {BEFORE | AFTER | INSTEAD OF} {event [OR ...]}
   ON table_name
   [FOR [EACH] {ROW | STATEMENT}]
   EXECUTE PROCEDURE trigger_function
```

  - – The event could be INSERT, UPDATE, DELETE or TRUNCATE.
  - – (BEFORE) or after (AFTER) event specifies the order of the trigger and the update
  - – INSTEAD OF is used only for views
  - – two kinds of triggers: row level trigger and statement level trigger,

Trigger Example – Referential constraint

- ❑ Figure 5.8 in DSC book does NOT work
  - – Rollback is not allowed in a trigger



❑ Figure 5.8 in DSC book:

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
            select time_slot_id
            from time_slot)) /* time_slot_id not present in time_slot */
begin
   rollback
end;
```

Will not work in MySQL
Roolback is not allowed

- ❑ The following is an alternative
  - – The result is the same: an update with a time_slot_id not present in the time_slot table will not be allowed (and will thus be ignored)

```
create function timecheck() -- the trigger function
returns trigger as $$
begin
  if (new.time_slot_id not in (select time_slot_id from time_slot)) then
    raise exception 'time_slot_id is unknown';
  end if;
end; $$
language plpgsql;

create trigger timecheck_trig -- the trigger (calling the trigger function
  before insert on section
  for each row execute procedure timecheck();
```

```
uni=# insert into section values
uni-# ('BIO-301', '2', 'Winter', '2009', 'Painter', '514', 'I');
ERROR:  time_slot_id is unknown
CONTEXT:  PL/pgSQL function timecheck() line 4 at RAISE
```

- Notice SQL and PostgreSQL details
  - The example is a **before** rather than an **after** trigger
  - **if** (inside the block) replaces **when** (outside)
  - "**referencing new row as** *nrow*" won't work, but you can reference the new value simply with **new**
    - **new** can be used in **insert** and **update**-triggers
    - **old** can be used similarly in **delete** and **update**-triggers
  - `raise exception` will prevent the insert and return an error message

```
create function timecheck() -- the trigger function
returns trigger as $$
begin
  if (new.time_slot_id not in (select time_slot_id from time_slot)) then
    raise exception 'time_slot_id is unknown';
  end if;
end; $$
language plpgsql;

create trigger timecheck_trig -- the trigger (calling the trigger function
  before insert on section
  for each row execute procedure timecheck();
```

```
uni=# insert into section values
uni-# ('BIO-301', '2', 'Winter', '2009', 'Painter', '514', 'I');
ERROR:  time_slot_id is unknown
CONTEXT:  PL/pgSQL function timecheck() line 4 at RAISE
```

Trigger Example – Ad hoc constraint

- Company policy (insert on instructor trigger)
  - No new employments in high budget departments (>=90000)
  - New employees (instructors) must never have a salary greater than everybody else

```
drop trigger if exists instructorcheck_trig on section;
drop function if exists instructorcheck;
create function instructorcheck() -- the trigger function
returns trigger as $$
begin
  if (new.dept_name not in (select dept_name from department where budget <90000)) th
    raise exception 'No no no, no new employees in the % department', new.dept_name;
  end if;
  if (new.salary> (select max(salary) from instructor)) then
    raise exception 'No no no, salary too high';
  end if;
end;$$
language plpgsql;

create trigger instructorcheck_trig -- the trigger
  before insert on instructor for each row execute procedure instructorcheck();
```

```
uni=# insert into instructor values (12345, 'Wong', 'Finance', 80000);
ERROR:  No no no, no new employees in the Finance department
```

```
uni=# insert into instructor values (23456, 'Wang', 'History', 100000);
ERROR:  No no no, salary too high
```

Triggering Events and Actions in SQL

- ❑ Triggering event can be **insert**, **delete** or **update**
- ❑ Triggers can be activated before an event, which can serve as extra constraints. E.g. convert blank grades to null.

```
create function setnull() -- the trigger function
returns trigger as $$
begin
       if (new.grade = '  ') then
               new.grade := null;
       end if;
       RETURN NEW;
end;$$
language plpgsql;

create trigger setnull_trig -- the trigger
  before update on takes for each row execute procedure
setnull();
```

Tr

```
create function setnull() -- the trigger function
returns trigger as $$
begin
        if (new.grade = ' ') then
                new.grade := null;
        end if;
        RETURN NEW;
end;$$
language plpgsql;

create trigger setnull_trig -- the trigger
  before update on takes for each row execute procedure
setnull();
```

testing …

```
uni=# select * from takes where grade is null;
  id   | course_id | sec_id | semester | year | grade
-------+-----------+--------+----------+------+-------
 98988 | BIO-301   | 1      | Summer   | 2010 |
(1 row)

uni=# update takes
uni-# set grade=' ' where id ='98765' and course_id='CS-101';
UPDATE 1
uni=# select * from takes where grade is null;
  id   | course_id | sec_id | semester | year | grade
-------+-----------+--------+----------+------+-------
 98988 | BIO-301   | 1      | Summer   | 2010 |
 98765 | CS-101    | 1      | Fall     | 2009 |
(2 rows)
```

Trigger to Maintain credits_earned value

- **Figure 5.9 from the DSC book**
  - ~~create trigger credits_earned after update of takes on (grade)~~

```
~~create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> ’F’ and nrow.grade is not null
    and (orow.grade = ’F’ or orow.grade is null)
begin atomic
     update student
     set tot_cred= tot_cred +
           (select credits
            from course
            where course.course_id= nrow.course_id)
     where student.id = nrow.id;
end;~~
```

Will not work in PostgreSQL

Trigger to Maintain credits_earned value (Cont.)

❑ **Figure 5.9 from the DSC book (New PostgreSQL version)**

```
create function credits_earned() -- the trigger function
returns trigger as $$
begin
  if (new.grade <> 'F' and new.grade is not null
    and (old.grade = 'F' or old.grade is null)) then
    update student
    set tot_cred = tot_cred +
      (select credits from course
       where course.course_id= new.course_id)
    where student.id = new.id;
  end if;
end;$$
language plpgsql;

create trigger credits_earned_trig -- the trigger
  after update on takes for each row execute procedure
credits_earned();
```

- **NOTICE: “update of** ***takes*** **on** (*grade*)” is not supported in Postgres
- **But we can simply use “**`update on takes`”

```
create function credits_earned() -- the trigger function
returns trigger as $$
begin
  if (new.grade <> 'F' and new.grade is not null
    and (old.grade = 'F' or old.grade is null)) then
    update student
    set tot_cred = tot_cred +
      (select credits from course
       where course.course_id= new.course_id)
    where student.id = new.id;
  end if;
end;$$
language plpgsql;

cr

cr
```

testing …

```
uni=# select * from student where id = '98988';
  id   |  name  | dept_name | tot_cred
-------+--------+-----------+----------
 98988 | Tanaka | Biology   |      120
(1 row)

uni=# update takes
uni-# set grade='C+' where id ='98988' and course_id='BIO-
301';
UPDATE 1
uni=# select * from student where id = '98988';
  id   |  name  | dept_name | tot_cred
-------+--------+-----------+----------
 98988 | Tanaka | Biology   |      124
(1 row)
```

Trigger example

- ❑ **An update** trigger ensuring that amount on account always satisfies 0 ≤ amount ≤ 100

```
create function upd_check() before update on account
returns trigger as $$
begin
   if new.amount < 0 then
      set new.amount = 0;
   elseif new.amount > 100 then
      set new.amount = 100;
   end if;
   return new;
end;

create trigger upd_check_trig -- the trigger
  after update on takes for each row execute procedure
upd_check();
```

Statement Level Triggers

- ❑ Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction.
  - – Can be more efficient when dealing with SQL statements that update a large number of rows
- ❑ **Supported by** some DBMS' (including **Postgres**) using
  - – **for each statement** instead of **for each row**

- ❑ **Insertion of 887000 rows:**

```
insert into movie.movie
  select id, title, production_year
  from imdb_movie.movie
  where kind_id=1;
```

- ❑ **with row-level: 887000 actions, with statement level: 1 action**