CIT Complex IT Systems
Section 1

Database Programming

Troels Andreasen

Outline

- Accessing SQL From a Programming Language, using API's, Exemplified
  - JDBC, ODBC and ADO.NET
    - ADO.NET with C# will be covered in more detail in CIT Section 2
- Programming the database
  - Functions and Procedural Constructs in SQL
  - Triggers in SQL
- Recursion in SQL

Accessing SQL from a Programming Language

A database programmer must have access to a general-purpose programming language for at least two reasons

- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language.
- Non-declarative actions -- such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface -- cannot be done from within SQL.

JDBC and ODBC and ADO.NET

- API's (application-program interfaces) 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 the CIT course
    - In particular we will consider what's called Object Relational Mapping (ORM) with Entity-Framework and using LINQ in C#
      - More about this in section 2

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.
- Approach for communicating with the database:
  1) Open a connection
  2) Create a “statement” object
  3) Execute query
  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{

      Class.forName("org.postgresql.Driver");

      // 1) Open a connection
      System.out.println("Connecting to database...");
      conn = DriverManager.getConnection(DB_URL,USER,PASS);

      // 2) Create a "statement" object
      System.out.println("Creating statement...");
      stmt = conn.createStatement();
      String sql;
      sql = "SELECT id, name, salary FROM instructor";
      // 3) Execute query
      ResultSet rs = stmt.executeQuery(sql);

      // 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"));
      }
```

```
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!
```

```
      // 5) Close connection
         rs.close();
         stmt.close();
         conn.close();
      }catch(Exception e){
         //Handle errors
         e.printStackTrace();
      }
      System.out.println("Goodbye!");
}//end main
}//end FirstExample
```

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
- Overall the approach for communicating with the database is the same:
  1) Open a connection
  2) Create a “statement” object
  3) Execute query
  4) Extract data from result set
  5) Close connection

C# & ADO.NET

```
using System;
using Npgsql;

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

            using (var conn = new NpgsqlConnection(connString))
            {
                // 1) Open a connection
                conn.Open();
                // 2) Create a “statement” object
                using (var cmd = new NpgsqlCommand("SELECT id, name, salary FROM instructor", conn))
                // 3) Execute query
                using (var rdr = cmd.ExecuteReader())
                // 4) Extract data from result set
                while (rdr.Read()){
                    Console.WriteLine("ID: {0} {1} {2}", rdr[0], rdr[1], rdr[2]);
                }
                // 5) Close the connection
                conn.Close();
            }
        }
    }
}
```

```
ID: 10101 Srinivasan 65000.00
ID: 12121 Wu 90000.00
ID: 15151 Mozart 40000.00
ID: 22222 Einstein 95000.00
ID: 32343 El Said 60000.00
ID: 33456 Gold 87000.00
ID: 45565 Katz 75000.00
ID: 58583 Califieri 62000.00
ID: 76543 Singh 80000.00
ID: 76766 Crick 72000.00
ID: 83821 Brandt 92000.00
ID: 98345 Kim 80000.00
```

Security issue – SQL Injection (to be covered later)

- Basically, communicating with the database goes like this
  - Build a query string like
    - "**select * from instructor where name = 'Srinivasan**'"
  - from
    - The string "**select * from instructor where name = '<input goes here>**'"
    - and a value like **Srinivasan** received as input
  - Send this to the database and get the result in return
- One important security issue, to be covered later: **SQL Injection**
  - To interfere with the construction of the query string
  - example input to do injection:
    - **X' or 'Y' = 'Y**
  - a trick to get all data from all instructors

Functions and Procedures

Functions and Procedures

- Functions and procedures allow **"business logic"** to be **stored in the database** and executed from SQL statements as well as called from application programs.

- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, Python, R, and others.

- The syntax presented in the DB book is defined by the SQL standard.
  - Most databases implement nonstandard versions of this syntax.

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 SQL 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, develop a 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 one of the most advanced frameworks and language extensions for adding functions and procedures to the DBMS

- **PL/pgSQL**
  - PL/pgSQL is the PostgreSQL version of a **module language** (a procedural programming language)
  - to be used for developing functions, procedures and triggers
  - PL/pgSQL was originally inspired by Oracle's PL/SQL language.

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**, that returns a table/relation as a result – just as any SQL query does.

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

- Many database systems have proprietary procedural extensions to SQL that differ from the standard.

SQL Functions

- Define a function.

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

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

Why is $$ needed?

- ";" is the end-of-statement symbol
- not enclosing the function body in $$ would make ";" ambiguous

$$ is used to enclose the body as a litteral

- Use the function.

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

```
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)
language plpgsql as
$$
begin
return concat('hello, ',s,'!');
end;
$$;
```

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 5.6

- 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
language plpgsql as $$
declare d_count integer;
begin
        select count(*) into d_count
        from instructor
        where instructor.dept_name = d_name;
        return d_count;
end;
$$;
```

- You can call the dept_count()-function to get a value:

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

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

```
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
```

Standard (DSC book) vs PL/pgSQL

```
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 5.6

Declarations outside block

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

SQL Procedures

- The *dept_count* function could instead be written as a 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 200

- Procedures can be called from
  - other procedures or
  - SQL embedded in application programs or
  - command line, using the **call** statement.

SQL Procedures

- The *dept_count* function could instead be written as a 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 200

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

SQL Procedures

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

calling the procedure from an anonymous code block (DO-block)

```
DO $$
declare
  n integer;
begin
  call dept_count_proc ('Comp. Sci.', n);
       RAISE NOTICE 'Comp. Sci. has % instructors', n;
end $$;
```

Output from RAISE NOTICE

```
NOTICE:  Comp. Sci. has 3 instructors
DO
uni=#
```

- Nothing is gained from using the procedure here
  A more straightforward coding would be a function as before:

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

PL/pgSQL Function

As before, with minor change

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

PL/pgSQL Procedure

SQL Functions

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

PL/pgSQL Function

- or an even simpler version as a plain SQL language function

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

SQL Function

SQL Procedures

- So, why at all consider stored procedures?
- One important argument:
  - user-defined functions cannot execute transactions
  - stored procedures can do this
- Example: Transferring an amount from one account to another

```
drop table if exists accounts;
create table accounts (
    id int generated by default as identity,
    name varchar(100) not null,
    balance dec(15,2) not null,
    primary key(id)
);
insert into accounts(name,balance) values('Bob',10000);
insert into accounts(name,balance) values('Alice',10000);
```

```
uni=# select * from accounts;
 id | name  | balance
----+-------+----------
  1 | Bob   | 10000.00
  2 | Alice | 10000.00
(2 rows)
```

SQL Procedures

- So, why at all consider stored procedures?
- One important argument:
  - user-defined functions cannot execute transactions
  - stored procedures can do this
- Example: Transferring amount from one account to another

```
drop table if exists accounts;
```

```
create procedure transfer(sender int,receiver int,amount dec)
language plpgsql
as $$
begin
    -- subtracting the amount from the sender's account
    update accounts
    set balance = balance - amount
    where id = sender;
    -- adding the amount to the receiver's account
    update accounts
    set balance = balance + amount
    where id = receiver;
    commit;
end;$$
```

SQL Procedures

- So, why at all consider stored procedures?
- One important argument:
  - user-defined functions cannot execute transactions
  - stored procedures can do this
- Example: Transferring amount from one account to another

```
drop table if exists accounts;
c
create procedure transfer(sender int, receiver int, amount dec)
language plpgsql
as $$
begin
    -- subtracting the amount from the sender's account
)
i
    update accounts
i
    set balance = balance - amount
    where id = sender;
    -- adding the amount t
    update accounts
    set balance = balance
    where id = receiver;
    commit;
end;$$
```

```
uni=# call transfer(1,2,500);
```

```
uni=# select * from accounts;
 id | name  | balance
----+-------+-----------
  1 | Bob   |  9500.00
  2 | Alice | 10500.00
(2 rows)
```

Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called **table functions**
- Example: Return all instructors in a given department

```
create function instructor_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 = instructor_of.dept_name);
```

DSC Figure 5.7

- Usage

```
select *
from table (instructor_of ('Music'))
```

Table Functions

- Same function, here using plain SQL (in the body)

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

- Here used to retrieve instructors in the Physics department.

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

Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.

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

- Loops using (among other) while and repeat statements:
  - **while** boolean expression **do**
    sequence of statements ;
    end while

  - **repeat**
    sequence of statements ;
    until boolean expression
    end repeat

Language Constructs – if-then-else

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

if *boolean expression*
then *statement or compound statement*
elseif *boolean expression*
then *statement or compound statement*
else *statement or compound statement*
end if

SQL Procedure, example with IF and WHILE loop

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

a table, **foo**, for testing

defining the **load_foo()** procedure

calling and showing the effect

```
create or replace procedure load_foo()
language plpgsql as
$$
declare
  i_max integer := 4;
  i integer := 0;
  n integer;
begin
  while i < i_max loop
    n:=(random() * 10000);
    if n>5000 then n=0;
    end if;
    insert into foo (val) values (n);
    i:=i+1;
  end loop;
end
$$;
```

```
uni=# call load_foo();
CALL

uni=# select * from foo;
 id | val
----+------
  9 |    0
 10 |    0
 11 | 4978
 12 |    0
(4 rows)
```

SQL Procedure, example with IF and WHILE loop

- 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 used to generate number between 0 and 9999
    - `n:=(random() * 10000);`
  - if statement to replace numbers > 5000 with 0
    - `if n>5000 then n=0;`

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 procedure test()
language plpgsql as
$$
begin
  call load_foo();
  call load_foo();
end
$$;
```

calling and showing the effect

```
uni=# call test();
CALL

uni=# select * from foo;
 id | val
----+------
 13 |    0
 14 | 2068
 15 | 4190
 16 | 2770
 17 |    0
 18 |  211
 19 | 2982
 20 |    0
(8 rows)
```

SQL Procedure, same functionality, but simplified

```
create or replace procedure load_foo()
language plpgsql as
$$
declare
  i_max integer := 4;
  i integer := 0;
  n integer;
begin
  while i < i_max loop
    n:=(random() * 10000);
    if n>5000 then n=0;
    end if;
    insert into foo (val) values (n);
    i:=i+1;
  end loop;
end
$$;
```

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

Cursor

- declared by a query
- supports row-by-row traversal of the result of the 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 …;`
- **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 procedure curdemo()
language plpgsql as
$$
declare
  rec record;
  cur1 cursor for select name,salary from instructor;
begin
  open cur1;
  loop
    fetch cur1 into rec;
    exit when not found;
    if rec.salary > 81000 then
      insert into vip
          values (rec.name,rec.salary);
    end if;
  end loop;
  close cur1;
end;
$$;
```

calling and showing the effect

```
uni=# call curdemo();
CALL

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 four “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 … to replace a normal cursor

- A very useful loop in PostgreSQL is the following

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

- Testing here with a **DO-block**
  - anonymous function
  - can be used for adhoc tasks and for testing expressions

- raise notice …
  - a kind of print() statement
  - can be used while testing

the FOR … IN loop provides a kind of "implicit" cursor

testing with a DO block

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

SQL using cursor (now implicit), example

a table, **vip**, for testing

```
-- vip(name, salary)
-- see slide 43
```

```
create or replace procedure curdemo2() 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=# call curdemo2();
CALL
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 examples above (slide 43 and slide 46)
  - 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 still a cursor

External Language Routines

- SQL allows us to define functions in a programming language such as Java, C, Python, R, and others.
  - Can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL can be executed by these functions.

- Declaring external language procedures and functions
  Examples:

```
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 5 languages:
  - SQL, PL/pgSQL, PL/Tcl, PL/Perl and PL/Python
- You can install other procedural languages
  - e.g., Java, Lua, R, Unix Shell, JavaScript, C++, …

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
  - There are alternatives, which give good security at the cost of potentially worse performance.

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, we must:
  - Specify the **conditions** under which the trigger is to be executed.
  - Specify the **actions** to be taken when the trigger executes.
- Triggers were introduced to the SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - For example, **after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as** : for inserts and updates

- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
      when (nrow.grade = ' ')
      begin atomic
         set nrow.grade = null;
      end;
```

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 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

- 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 5.9 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 PostgreSQL

Trigger Example – Referential constraint

- Figure 5.9 in DSC book does NOT work
  - Rollback is not allowed in a trigger in PostgreSQL
- 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)

- Figure 5.9 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 PostgreSQL

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

% is a placeholder to be replaced by the value of **new.dept_name**

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

```
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   | 2018 |
(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   | 2018 |
 98765 | CS-101    | 1      | Fall     | 2017 |
(2 rows)
```

Trigger example SKAL RETTES

- **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();
```

Trigger to update tot_cred on student

- **Keeping the value of tot_cred up-to-date**

```
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;
```

DSC Figure 5.10

Trigger to update tot_cred on student

- Keeping the value of tot_cred up-to-date
- In PostgreSQL with
  - a trigger function and
  - a trigger

```
create function update_tot_cred() -- the trigger function
returns trigger as $$
begin
        update student
        set tot_cred=(select sum(credits)
                from takes join course using (course_id)
                where takes.id=student.id)
        where student.id=new.id;
        return NEW;
end; $$
language plpgsql;

create trigger update_tot_cred_trig -- the trigger
  after insert on takes
  for each row execute procedure update_tot_cred();
```

Trigger to update tot_cred on student

- Keeping the value of tot_cred up-to-date
- In PostgreSQL with
  - a trigger function and
  - a trigger
- Testing

```
uni=# select * from student where id='19991';
  id   |  name  | dept_name | tot_cred
-------+--------+-----------+----------
 19991 | Brandt | History   |       80
(1 row)

uni=# insert into takes values ('19991', 'CS-190', '2', 'Spring', '2017', 'A');
INSERT 0 1
uni=# select * from student where id='19991';
  id   |  name  | dept_name | tot_cred
-------+--------+-----------+----------
 19991 | Brandt | History   |        7
(1 row)
```

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
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows

- **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**

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - Maintaining **summary data** (e.g., total salary of each department)
  - **Replicating databases** by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication

- Encapsulation can be used instead of triggers in many cases
  - Define methods og SQL procedures to update fields
  - Carry out actions as part of these