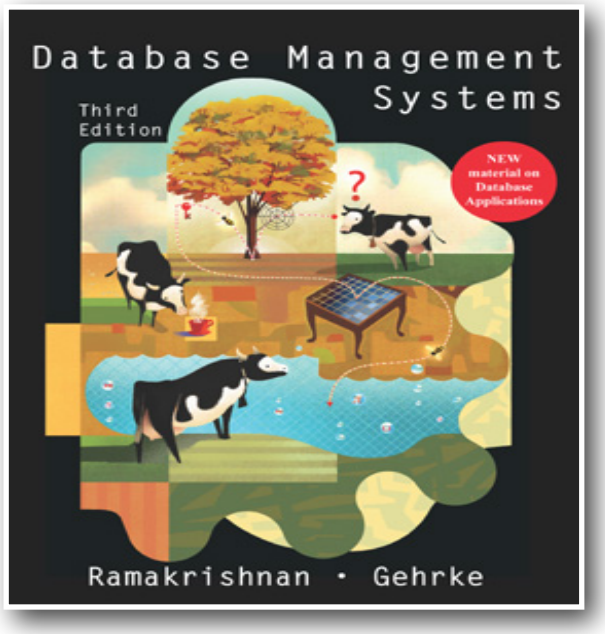
TÉCNICO LISBOA

Sistemas de Informação e Bases de Dados

Class 11: Triggers e Stored Procedures

Prof. Paulo Carreira

Bibliography



Chapters 3 e 5

POSTGRES Manual

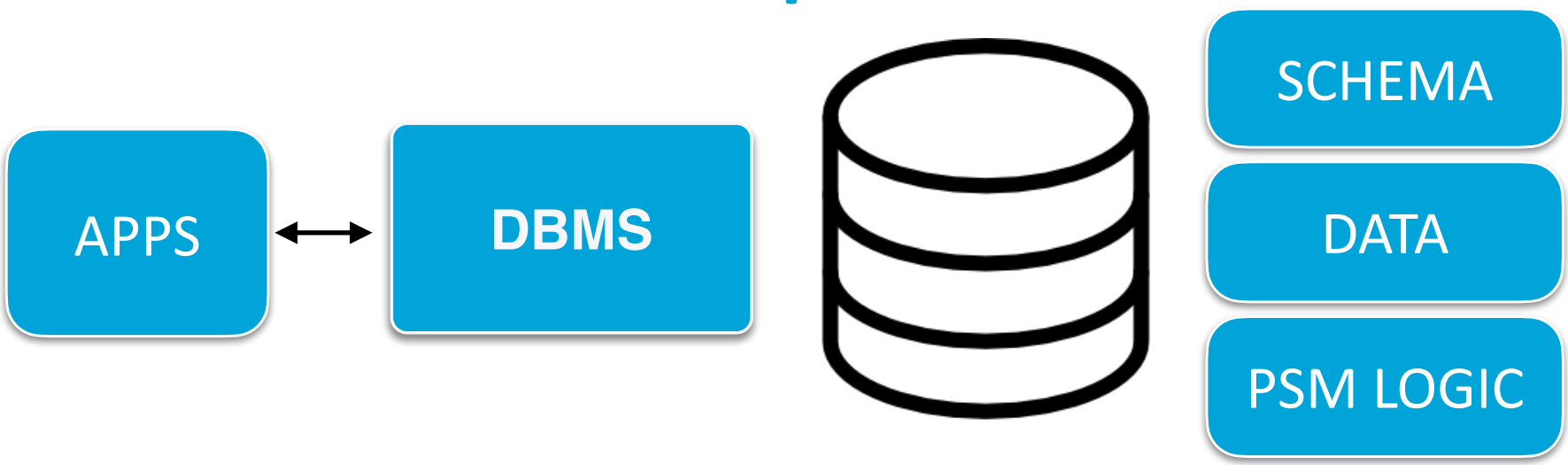
- https://www.postgresql.org/docs/9.5/static/sql-createtrigger.html
- https://www.postgresql.org/docs/9.5/static/plpgsql-structure.html
- https://www.postgresql.org/docs/9.5/static/plpgsql-control-structures.html

Class Outline

- Introduction to PSM
- Stored Functions and Procedures
- Syntax of the main block elements
- Cursors

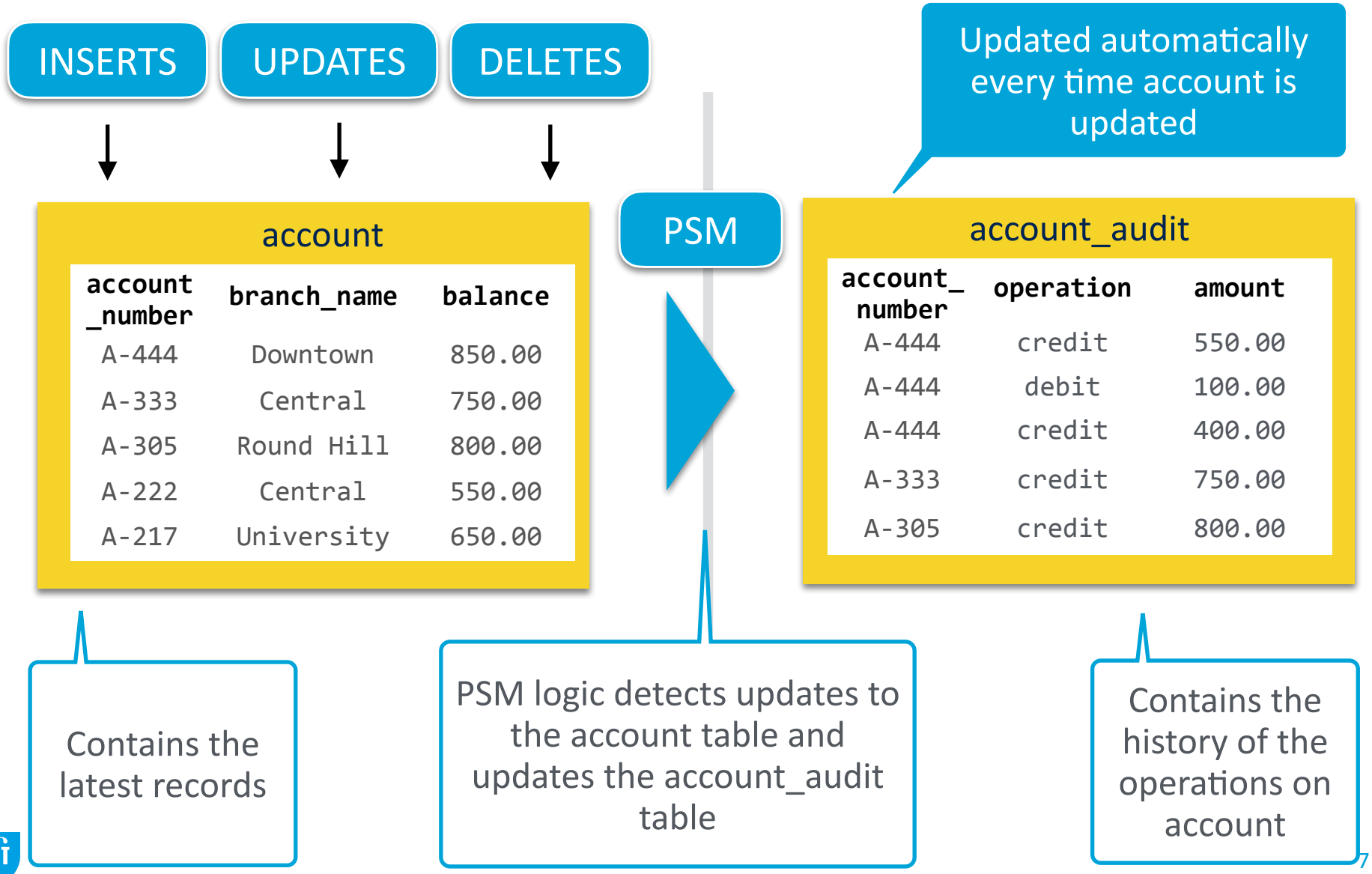
Introduction

Data Centric Development



Data-centric database development is a development approach whereby procedural instructions (like as a sub-program in any programming language) can be stored inside a DB also known as a stored program / stored routine

Motivation Example



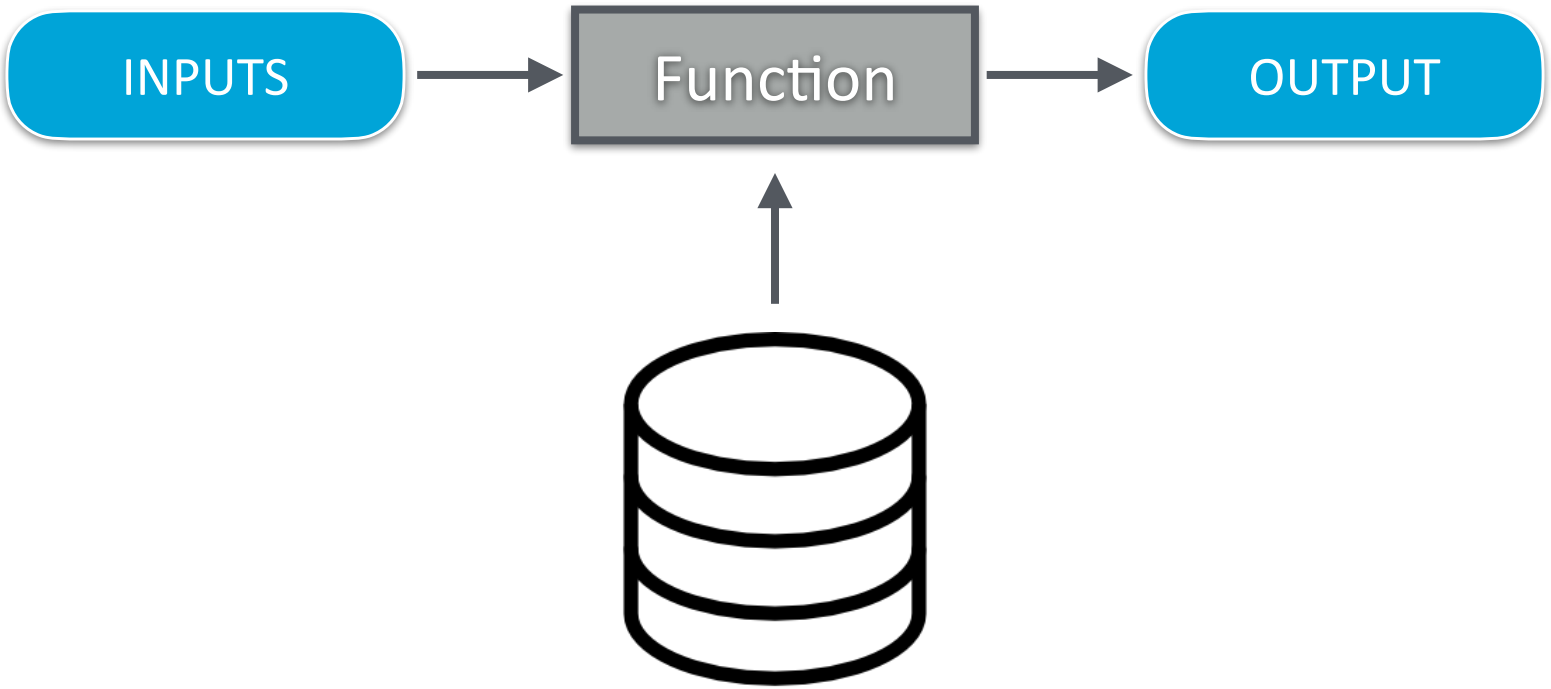
Persistent Stored Modules

PSM

A Persistent Stored Module can be of two kinds:

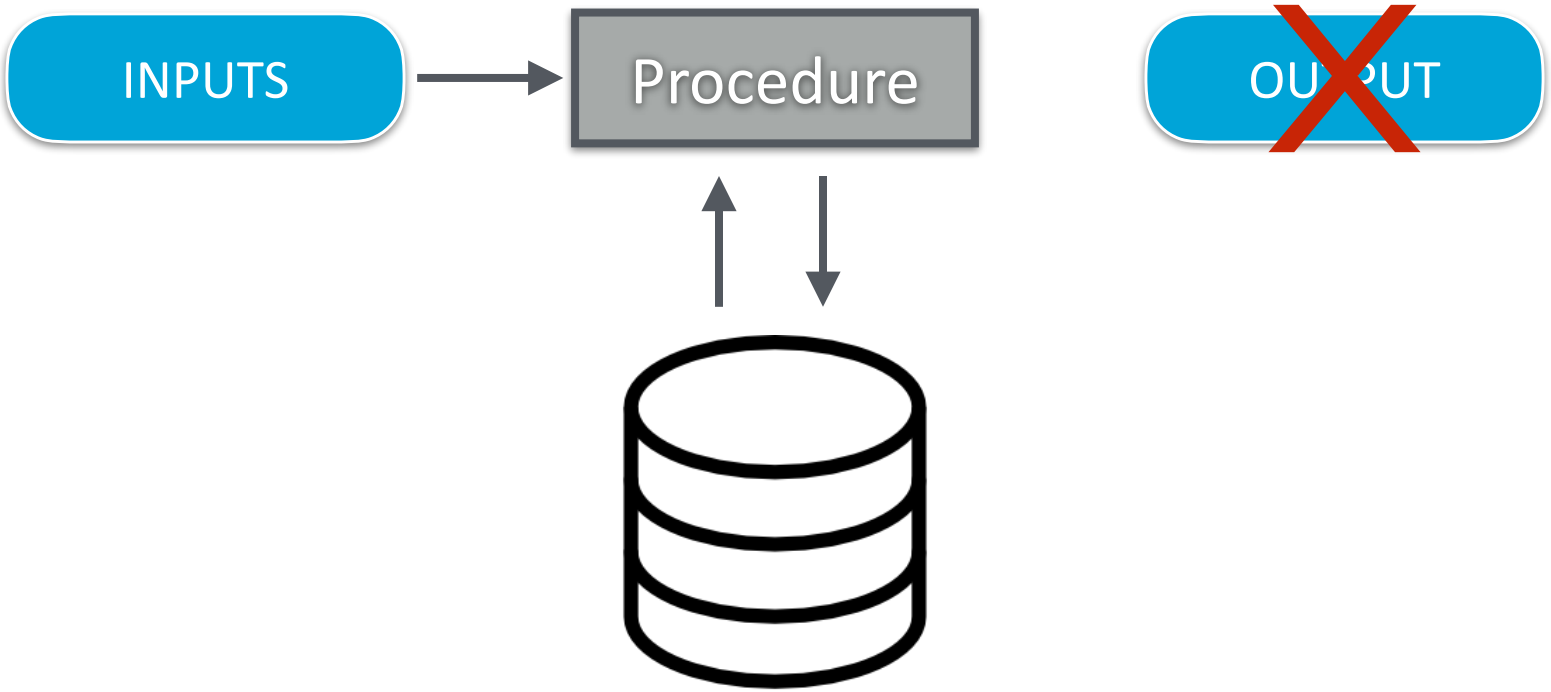
- A stored *function* that returns a value but that should not change the state of the database.
- A *stored procedure* that may change the state of the database and does not return a value

Stored Function



A **Stored Function** combines inputs and results of queries from the database to produce the output. The database is never changed.

Stored Procedure



A **Stored Procedure** combines inputs and results of queries from the database to produce the output. The database is never changed.

RDBMS procedural languages

- (*c.* 1992) **PL/SQL** – Oracle
- (*c.* 1998) **PL/pgSQL** – Postgres (≈PL/SQL)
- (*c.* 1999) **SQL/PSM** – Standard SQL (IBM DB2 and MySQL)
- (*c.* 1989) **Transact-SQL** – MS SQL Server + Sybase

Advantages

- Can make applications faster: PSM are compiled and maintained inside the database
- Reduce the exchange of data: especially between the application and the DB server
- Introduce a level of indirection: can be called from applications written in distinct languages

Drawbacks

Development Complexity: Maintaining PSMs is relatively complex (especially when domain logic is complex)

- Debugging and profiling become very difficult
- Development environment highly dependent on the DBMS used and on the toolchain of the manufacturer

Definition and invocation

- Invoking a procedure and a function

```
CREATE [OR REPLACE] PROCEDURE/FUNCTION …

DROP [OR REPLACE] PROCEDURE/FUNCTION …
```

- Defining a procedure or a function

```
CALL nome_procedimento [(param1, param2,…)]

my_function_name([param1, param2,…])
```

Stored functions and procedures can be called from other procedures, or from client applications written in C, PHP, Java, Javascript

Statements
(the body of a PSM)

Example procedure (PSM Standard)

```
CREATE PROCEDURE my_procedure(IN parameter1 INTEGER) AS

DECLARE variable1 CHAR(10);
BEGIN
     IF parameter1 = 17
     THEN
        SET variable1 := 'birds';
     ELSE
        SET variable1 := 'beasts';
     END IF;
     INSERT INTO table1
         VALUES (variable1);
END
```

Name and parameters

Variable declarations

Start of the statement block

Assignment

SQL statement

Stored Functions in POSTGRES

Language Token separator

```
CREATE FUNCTION myfunc([params])
RETURNS type AS
$$
DECLARE [declarations]
BEGIN
  [statements]
END
$$ LANGUAGE plpgsql;
```

PL/PgSQL does not support true stored procedures; only **stored functions** are supported.

CREATE FUNCTION myfunc(*[params]*) **RETURNS VOID AS**

PSM in Postgres does not follow the SQL PSM Standard (although they are very similar)

The ‘Hello World’ of PSM

Define a simple function to add two numbers

```
CREATE FUNCTION add_me(
    x NUMERIC,
    y NUMERIC)
    RETURNS NUMERIC AS
$$
BEGIN
  RETURN x + y;
END
$$ LANGUAGE plpgsql;
```

Executing a Stored Function

Distinct ways to execute a stored function

- Direct query (without FROM)

```
SELECT add_me(2,3);
```

```
 add_me
--------
      5
```

- As an expression in the SELECT (applied to every line)

```
SELECT name, add_me(salary, 100)
FROM employee
```

- As an expression in the WHERE (applied to every line)

```
SELECT *
FROM employee
WHERE add_me(salary, -1000) < 5000
```

Specifying Deterministic Functions

Definition

A function is *deterministic* if it always produces the same results for the same input values

```
CREATE FUNCTION myfunc([params]) RETURNS type
  IMMUTABLE
```

- Example of a non-deterministic function: invokes the `NOW()` or `RAND()` function
- In POSTGRES, every functions are non-deterministic by default (we must specify **IMMUTABLE** for deterministic functions)
- Immutable (i.e. deterministic) functions can be optimised

Variable Declarations

```
DECLARE var_name [, var_name ...] type
   [DEFAULT value]
```

Variables are only visible in the scope of the BEGIN . . . END block where they were declared

```
DECLARE
   first_name VARCHAR(50) DEFAULT 'John';
   last_name  VARCHAR(50) DEFAULT 'Doe';
   counter    INTEGER := 1;
   payment    NUMERIC(11,2) := 20.5;
```

Example

A function that, given the name of a depositor, returns how many accounts that depositor has

```
CREATE OR REPLACE FUNCTION account_count(
    d_name VARCHAR(40))
RETURNS INTEGER AS
$$
DECLARE total_count INTEGER;
BEGIN
  SELECT COUNT(*) INTO total_count
   FROM depositor
   WHERE customer_name = d_name;

  RETURN total_count;
END
$$ LANGUAGE plpgsql;
```

Example Usage

Get the name, street and city of customers with more than one account

```
SELECT
    customer_name,
    customer_street,
    customer_city
FROM customer
WHERE account_count(customer_name) > 1
```

Demonstration example only.
Do not use in practice!

Assigning values to local variables

- Assigning values from expressions

```
var_name := expr
```

POSTGRES syntax

```
LET var_name := expr
```

```
SET var_name := expr
```

Other RDBMS

- Capturing the result of query

```
SELECT col_name, ...
INTO var_name
FROM …
```

Conditionals and Cycles

```
IF condition THEN statement_list1
ELSE statement_list2
END IF;

WHILE condition
DO statement_list
END WHILE;

REPEAT statement_list
UNTIL condition
END REPEAT;

LOOP statement_list
END LOOP;
```

Examples

A PSM that returns a table

A function that, given the name of the customer, returns all customer accounts

```
CREATE FUNCTION accounts_of(name VARCHAR(80))
 RETURNS SETOF account
AS
$$
 SELECT a.account_number, branch_name, balance
  FROM account as a, depositor as d
 WHERE a.account_number = d.account_number
       and d.customer_name = name;
$$
LANGUAGE sql;
```

Function with complex types

Function that, given the name of a customer, returns multiple rows (a table) with the customer's accounts

```
DROP TYPE account_data;

CREATE TYPE account_data AS (
   account_number  CHAR(5),
   branch_name     VARCHAR(80),
   balance         NUMERIC(16,4)
);
```

```
DROP FUNCTION accounts_of(varchar);

CREATE FUNCTION accounts_of(name VARCHAR(80))
    RETURNS account_data
AS $$
   SELECT account_number, branch_name, balance
   FROM account NATURAL JOIN depositor
   WHERE customer_name = name;
$$ LANGUAGE sql;
```

```
SELECT * FROM accounts_of('Cook');
```

```
SELECT accounts_of('Cook'); -- Works but gives different result
```

A PSM that returns a row of a table

```
CREATE OR REPLACE FUNCTION accounts(
    acct_num VARCHAR(10))
RETURNS account AS
$$
DECLARE
    acct account%ROWTYPE;
BEGIN
    SELECT *
    INTO acct
    FROM account
    WHERE acct_number = acct_num;

    RETURN acct;
END
$$
LANGUAGE plpgsql;
```

Syntax for "Stored Procedures"

```
CREATE FUNCTION myfunc([params]) RETURNS type
```

- Only have stored functions are supported in POSTGRES: The **RETURNS** statement must be specified

- Procedures are functions that do no return anything: Obtained by specifying **RETURNS VOID**

https://www.postgresql.org/docs/9.2/static/sql-createfunction.html

DO blocks

DO executes an anonymous code block

```
DO $$
DECLARE total NUMERIC DEFAULT 0;
BEGIN
    SELECT SUM(balance)
    INTO total
    FROM account;

    RAISE INFO 'Account total --> %', total;
END$$;
```

```
INFO:  Account total --> 6300.0000
```

The code block is treated as though it were the body of a function with no parameters returning void

Cursors

Abstraction to read a table with a file semantics that can be used within Persistent Stored Modules

- **OPEN** - Open the cursor (and execute the query associated to the cursor)
- **FETCH** - Read the record and advance to the next
- **CLOSE** - Close the cursor and free resources

Cursors

⚠ Never use floating-point data types for money

```
CREATE OR REPLACE FUNCTION average_balance() RETURN FLOAT
   AS
$$
  DECLARE balance_var REAL DEFAULT 0.0;
  DECLARE sum_balance REAL DEFAULT 0.0;
  DECLARE count_balance INTEGER DEFAULT 0;
  DECLARE cursor_account CURSOR FOR
       SELECT balance FROM account;
BEGIN
  OPEN cursor_account;
  LOOP
    FETCH cursor_account INTO balance_var;
    sum_balance := sum_balance + balance_var;
    count_balance := count_balance + 1;
  END LOOP;
  CLOSE cursor_account;
  RETURN sum_balance / count_balance;
END
$$ LANGUAGE plpgsql;
```

Declares the cursor for a query

Opens the cursor

Reads a record and advances to the next record

Closes the cursor

TÉCNICO LISBOA

Sistemas de Informação e Bases de Dados

Class 12: Triggers, Stored Procedures, Views

Prof. Paulo Carreira

slides não são livros

Class Outline

- Exceptions
- Triggers
- Views

Exceptions

Raising Exceptions

RAISE EXCEPTION *<message>*
USING HINT '*text*'

```
CREATE FUNCTION search(uid INT) RETURNS VOID AS
$$
BEGIN
  IF NOT EXISTS(
            SELECT *
            FROM users
            WHERE user_id=uid)
  THEN
     RAISE EXCEPTION 'Nonexistent ID %', user_id
     USING HINT = 'Please check your user ID';
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Recovering from Exceptions

```
CREATE OR REPLACE FUNCTION calc_tax(arg NUMERIC) RETURNS
NUMERIC
AS $$
DECLARE res INTEGER;
BEGIN
  -- main block
  res := 100 / arg;
  RETURN res;

  EXCEPTION
     WHEN division_by_zero
     THEN RETURN 0;
END;
$$
LANGUAGE plpgsql;
```

Some code that triggers the exception

Use the WHEN clause to catch the exception

Triggers

Triggers

Are **actions** (procedures) that are automatically invoked in response to specific database **events** (operations) such as inserts, updates, or deletes

- Implement complex Integrity Constraints
- Update/change tables (for history or audit purposes) when some other table is changed

Triggers E-C-A pattern

Triggers are specified using *event-condition-action* (ECA) rules:

- **Event**: what type of update activates the trigger
- **Condition**: a question or test to see if the action should be taken
- **Action**: a procedure that is executed when the trigger is activated and the previous condition is true

Syntax

A procedure that is automatically invoked in response to specific database updates

- Creating a Trigger

```
CREATE TRIGGER <trigger_name>
   { BEFORE | AFTER } { INSERT | UPDATE | DELETE }

ON <tbl_name>

WHEN <condition>

FOR EACH { ROW | STATEMENT } EXECUTE PROCEDURE <proc_name>
```

- Removing a Trigger

```
DROP TRIGGER trigger_name
```

Triggers are an old concept was only standardised in the SQL standard: 1999

Example: Integrity Constraint

Create a trigger that prevents the balance from being negative or greater than 100

```
CREATE OR REPLACE FUNCTION chk_balance_interval_proc()
RETURNS TRIGGER AS
$$
BEGIN
  IF NEW.balance < 0 OR NEW.balance > 100 THEN
    RAISE EXCEPTION 'Withdrawal or deposit past the limits';
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

In POSTGRES we must create a function that returns a TRIGGER! 🤦

```
CREATE TRIGGER chk_balance_interval
BEFORE UPDATE OR INSERT ON account
FOR EACH ROW EXECUTE PROCEDURE chk_balance_interval_proc();
```

This trigger may fail

```
UPDATE account
SET balance = balance - 500
WHERE account_number = 'A-101';
```

Example: Integrity Constraint

Create a trigger that never lets the balance go below 0 or above 100

```
CREATE OR REPLACE FUNCTION chk_balance_interval_proc()
RETURNS TRIGGER AS
$$
BEGIN
  IF NEW.balance < 0 THEN
    NEW.balance := 0;
  ELSEIF NEW.balance > 100 THEN
    NEW.balance := 100;
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Code to install the trigger should go here…
```

This trigger never fails; instead, it intercepts the updates to balance

```
UPDATE account
SET balance = balance - 500
```

All account values will remain between 0 and 100

Trigger Failure Behaviour

- If a **BEFORE** trigger fails, the operation on the corresponding row or table is not performed
- An **AFTER** trigger is only executed if any **BEFORE** triggers on the same table and related to the same operation are successfully executed

Example: Complex table update

Consider the following requirements for the Bank database

- Whenever a customer withdraws more than the balance
- Instead of resulting in a negative balance, the bank:
  1. Creates a loan equal to the missing amount
  2. Gives the loan the same number as the account
  3. Zeroes the account balance
- **Trigger condition**: an update that results in a negative account balance

Example: Complex table update

Withdrawal of € 800 from the A-102 account

account_number	branch_name	balance
A-101	Downtown	500.0000
A-215	Metro	600.0000
A-102	Uptown	700.0000
A-305	Round Hill	800.0000
A-201	Uptown	900.0000
A-222	Central	550.0000
A-217	University	650.0000
A-333	Central	750.0000
A-444	Downtown	850.0000

Trigger: Example 2

Withdrawal of € 800 from the A-102 account result on a loan

Account

account_number	branch_name	balance
A-101	Downtown	500.0000
A-215	Metro	600.0000
A-102	Uptown	700.0000
A-305	Round Hill	800.0000
A-201	Uptown	900.0000
A-222	Central	550.0000
A-217	University	650.0000
A-333	Central	750.0000
A-444	Downtown	850.0000

Loan

loan_number	branch_name	amount
L-17	Downtown	1000.0000
L-23	Central	2000.0000
L-15	Uptown	3000.0000
L-14	Downtown	4000.0000
L-93	Metro	5000.0000
L-11	Round Hill	6000.0000
L-16	Uptown	7000.0000
L-20	Downtown	8000.0000
L-21	Central	9000.0000
A-102	UpTown	100.0000

Depositor

customer_name	account_number
Johnson	A-101
Brown	A-215
Cook	A-102
Cook	A-101
Flores	A-305
Johnson	A-201
Iacocca	A-217
Evans	A-222
Oliver	A-333
Brown	A-444

Borrower

customer_name	loan_number
Iacocca	L-17
Brown	L-23
Cook	L-15
Nguyen	L-14
Davis	L-93
Brown	L-11
Gonzalez	L-17
Iacocca	L-16
Parker	L-20
Brown	L-21
Cook	A-102

Trigger: Example 2

```
CREATE OR REPLACE FUNCTION overdraft_proc()
RETURNS TRIGGER
AS $$
BEGIN
   IF NEW.balance < 0 THEN
     INSERT INTO loan
     VALUES (NEW.account_number,
             NEW.branch_name,
             NEW.balance);

     INSERT INTO borrower
     SELECT customer_name, account_number
     FROM depositor
     WHERE depositor.account_number = NEW.account_number;

     NEW.balance := 0;
   END IF;
END
$$ LANGUAG
```

```
CREATE TRIGGER overdraft_trigger
BEFORE UPDATE ON account
FOR EACH ROW EXECUTE PROCEDURE overdraft_proc();
```

Problems with *Triggers*

- **Complex effect**. Its effect can be complex and unpredictable
  - *Multiple triggers* can be triggered in a single operation
  - The action of a trigger can activate another trigger (*recursive triggers*)
- **Cycles of events**. Chains of endless events. Very difficult to debug and fix.

Problems with Triggers

- **Unintended executions**
  - Changes to a table will run triggers even if that's not what is intended
- **Occurrence of errors**
  - If the trigger fails, the entire operation fails
  - Extended undo recovery times

When **not** to use Triggers

- Summary tables: Use *VIEWS* instead of triggers, if possible
- Complex Integrity Constraints: Use *CHECKs* whenever possible
- Table replication: Use DBMS build-in mechanisms

Views

Views

- A view is a *virtual table* defined through a query
- Associates a name to a **SELECT** statement

CREATE VIEW *myview* **AS SELECT** …

Once created, it can be used as a *relation* but it is not the same as creating a table

Views do not have storage, they are computed, and their contents will change if the tables involved in the query change

Creating Views

Views are database objects that can be *created* and *removed*.

- Creation of View

```
CREATE VIEW myview AS SELECT …
```

- Removal of View

```
DROP VIEW myview
```

Creating a View: Example 1

```
CREATE VIEW account_stats(name, num_accts)
AS
SELECT name, COUNT(*) AS num_accts
FROM depositor
GROUP BY customer_name;
```

```
SELECT * FROM account_stats;
```

Resolving Queries on Views

The technique for evaluating view queries is known as

View Expansion

```
SELECT COUNT(*)
FROM top_employee
WHERE department = ‘HR'
```

References to a view are replaced by its definition

```
SELECT COUNT(*)
FROM (SELECT name
   FROM employee
   WHERE salary > 10000)
WHERE name = 'Joaquim'
```

Views, data independence and security

Views can ***map*** data from tables to a new *logical model*.

- **Views** support logical independence from the *physical model*
- **Views** are useful for security context:
  - The DBA can views and grant them access to a group of users.

Updatable Views

SQL: 1999 distinguishes between two types of views:

- **Updatable Views**: Whose rows can be modified
  - A column of a view can be updated if it is obtained from exactly a base table and the primary key of the base table is included in the columns of the view
- **Insertable Views:** And views where new lines can be inserted
  - There must be a one-to-one relationship between the rows of the view and those of the respective base tables.

Updatable view: Example

This View is updatable

```
CREATE VIEW senior_employees(eid, emp_name, birthdate)
AS
    SELECT e.eid, e.name as emp_name, e.birthdate
    FROM employee e
    WHERE birthdate < '01-01-1980';
```

```
INSERT INTO senior_employees VALUES (7, 'Grace', '01-12-1979');
SELECT * FROM senior_employees;
```

```
UPDATE senior_employees
SET birthdate = '01-12-1989'
WHERE eid = 7;
```

```
SELECT * FROM senior_employees;
```

Non-updatable view: Example

This view is **not** updatable!

```
CREATE VIEW senior_managers(emp_name, dep_name, birthdate) AS
    SELECT e.name AS emp_name,
           d.name AS dep_name,
           e.birthdate
    FROM employee e
        JOIN department d ON e.eid = d.mid
WHERE birthdate < '01-01-1980';
```

```
SELECT * FROM senior_managers;
```

```
 emp_name | dep_name |  birthdate
----------+----------+------------
Caroline  | Sales    |  1971-04-07
```

This statement fails

```
INSERT INTO senior_managers VALUES ('Grace', 'Finance',
'08-03-1979');
```

Materialised Views

Views sometimes can be too costly to compute and need to be optimised

- ***Memory cache*** when the query underlying the view is complex and the results fit in memory
- ***Disk Cache*** when the query underlying the view is complex and the results do **not** fit in memory

- *How are updates to the underlying table propagated?*

Materialized Views

Large complex views can be materialised to optimise query performance (over those views)

```
CREATE MATERIALIZED VIEW account_stats AS SELECT …
```

- *Materialized query tables* and materialised *summary tables on IBM DB2*

```
CREATE TABLE account_stats AS SELECT …
```

Refreshing Materialized Views

Also known as *updating views* or *propagating updates*

In POSTGRES, views can be refreshed

- *ON-DEMAND*

```
REFRESH MATERIALIZED VIEW account_stats;
```

- *AUTOMATICALY*

```
CREATE UNIQUE INDEX idx_account_stats
ON account_stats(name);
```

```
REFRESH MATERIALIZED VIEW CONCURRENTLY
  account_stats;
```

Resolving Queries over Materialised Views

Queries over materialised views can be resolved using the *view expansion* strategy or using query rewriting over views

```
SELECT name, COUNT(*)
FROM depositor d
GROUP BY name
HAVING count(*) > 2
```

```
SELECT name, num_accts
FROM  account_stats
WHERE num_accts > 2
```

Detects that we are doing an aggregation query and rewrites the query at the expense of the view (or *existing* materialized views)