PLpgSQL

COMP9311 24T2; Week 4

By Zhengyi Yang, UNSW

Notice

- Deadline for Assignment 1 has passed
- Project released
- Deadline: 10PM, 7 July (Sunday Week 6)

SQL

SELECT

- Single Table
- Multiple Tables

Aggregation

- GROUP BY
- HAVING

Data definition

- CREATE TABLE

Modification

- INSERT/DELETE/UPDATE

Change schemas

- ALTER

Views

Can SQL do this?

Consider the **scenario:**

- Withdraw money at an ATM
- A bank customer attempts to withdraw funds in their account.
- An ATM interacts with a secure database with your banking details.

What can SQL do?(cont.)

Example: say a person with acctNum 1 is trying to withdraw 50 dollars

Imagine that this is the implementation for the bank withdraw scenario:

```
Select 'Insufficient Funds'
from Accounts
where acctNo = 1 and balance < 50;

Update Accounts
set balance = balance - 50
where acctNo = 1 and balance >=  50;

Select 'New balance:' || balance
from Accounts
where acctNo = 1;
```

What can SQL do?(cont.)

We can feel that it implicitly defines two evaluation scenarios:

- Display ‘Insufficient Funds’, UPDATE has no effect, displays unchanged balance
- UPDATE occurs as required, displays changed balance

i.e., *If there is not enough funds, the ATM should indicate ‘Insufficient Funds’; otherwise, it should allow the withdrawal and update the account balance.*

What can SQL do?(cont.)

```
Select 'Insufficient Funds'
from Accounts
where acctNo = 1 and balance < 50;

Update Accounts
set balance = balance - 50
where acctNo = 1 and balance >=
50;

Select 'New balance:' || balance
from Accounts
where acctNo = 1;
```

Some ***issues***:

1. There is no parameterization (e.g. acctNum, amount)
2. Will always attempt UPDATE, even when it knows it's invalid
3. Will always display a “new” balance, even if it’s unchanged

To accurately express the “business logic“ of withdrawing money, we need facilities like **conditional controls**.

The Limitation of SQL

What we have seen from SQL:

- Data definition (create table(...))
- Query (select...from...where...)
- Constraints on values (domain, key, referential integrity)

And some useful functionalities.

- Views (giving names to SQL queries)

However, this is not enough to support real applications. Therefore, more **extensibility** and **programmability** are needed.

SQL as a Programming Language

SQL is a powerful language for manipulating relational data, but it **is not meant to be a powerful programming language.**

What if at some point in developing complete database applications

- We will need to consider implementing user interactions
- we need to control sequences of database operations
- we need to process query results in additional ways

How would SQL be able to handle these?

Extending SQL by PostgreSQL

Ways that SQL could be extended:

- **new data types** (incl. constraints, I/O, indexes, ...)
- **more powerful constraint checking**
- **packaging/parameterizing queries**
- **more functions/aggregates for use in queries**
- **event-based triggered actions**

All are required to **assist application development**.

Database Programming

(Let’s return to the example of withdrawing money)

To return one of the two possible text results :

- If *try to withdraw too much* => return *'Insufficient funds’*
- If *withdrawal ok* => return *'New balance: newAmount'*

Requires a combination of

- **SQL code** to access the database
- **procedural code** to control the process

Database Programming

Database programming requires a ***combination*** of

- manipulation of data in DB   (via SQL)
- conventional programming   (via procedural code)

This combination is realized in several ways:

- Passing SQL commands via a ***"call-level" interface***
  (PL is decoupled from DBMS; most flexible; e.g., Java/JDBC, Python/ODBC)
- ***Embedding SQL*** into augmented programming languages
  (requires PL pre-processor; typically, DBMS-specific; e.g. SQL/C)
- …

A Stored Procedure Approach

Stored procedures

- procedures/functions that are **stored in DB** along with data
- written in a language combining SQL and **procedural** ideas
- provide a way to **extend** operations available in the database
- executed **within the DBMS** (close coupling with query engine)

Benefits of using stored procedures:

- minimal data transfer cost SQL ↔ procedural code
- user-defined functions can be nicely integrated with SQL
- procedures are managed like other DBMS data (ACID)
- procedures and the data they manipulate are held together

SQL/PSM

SQL/PSM is a **1996 standard for SQL** stored procedures. (PSM = Persistent Stored Modules)

Syntax for PSM procedure/function dentitions:

```
CREATE PROCEDURE ProcName ( <ParamList> )
[ local declarations ]
procedure body ;
```

```
CREATE FUNCTION FuncName (<ParamList> )
RETURNS Type
[ local declarations ]
function body ;
```

Parameters have three modes: IN, OUT, INOUT

Parameters

- **IN:** A variable passed in this mode is of **read-only** nature.
- **OUT:** In this mode, a variable is **write-only** and can be passed back to the calling program. It cannot be read inside the procedure and needs to be assigned a value.
- **INOUT:** This procedure has features of **both** IN and OUT mode. The procedure can also read the variables value and can also change it to pass it to the calling function.

SQL/PSM

Example: Defining a procedure:

```
CREATE PROCEDURE AddNewPerson (
IN name CHAR(20),
IN id INTEGER
)
INSERT INTO People VALUES(name, id);
```

Example: Invoking a procedure using the SQL/PSM statement CALL

```
CALL AddNewPerson(‘Codd’, 000001);
```

Status of PSM in Modern DB

Unfortunately, the PSM standard was ***developed after*** most DBMSs had their own stored procedure language -> **No** DBMS implements the PSM standard exactly.

1. IBM's DB2 and MySQL implement the SQL/PSM closely (but not exactly)
2. Oracle's PL/SQL is moderately close to the SQL/PSM standard
3. PostgreSQL's PLpgSQL is close to PL/SQL (95% compatible)

PostgreSQL

- We can pass SQL commands via a "call-level" interface
  (PL is decoupled from DBMS; most flexible; e.g., Java/JDBC, Python/ODBC)
- We can embed SQL into augmented programming languages
  (requires PL pre-processor; typically, DBMS-specific; e.g. SQL/C)
- Database programming can also be realized via special-purpose programming language **in the DBMS**
  - integrated with DBMS;
  - enables extensibility;
  - e.g. PL/SQL, PL/pgSQL.

User-defined Data Types

SQL data definition language provides:

- atomic types: integer, float, character, Boolean
- ability to define tuple types (create table)

PostgreSQL also provides mechanisms to define new types:

- basic types: ***CREATE DOMAIN***
- tuple types: ***CREATE TYPE***

User-defined Data Types(cont.)

Syntax for defining a new atomic type (as specialization of existing type):

```
CREATE DOMAIN DomainName [ AS ] DataType
[ DEFAULT expression ]
[ CONSTRAINT ConstrName constraint ]
```

Example

```
Create Domain UnswCourseCode as text
check ( value ~ '[A-Z]{4}[0-9]{4}' );
```

which can then be used like other SQL atomic types

```
Create Table Course (
    id integer,
    code UnswCourseCode, ...
);
```

~ is POSIX Regular Expressions

POSIX regular expressions provide a more powerful means for pattern matching than LIKE and SIMILAR TO.

User-defined Data Types(cont.)

Syntax for defining a new tuple type:

CREATE TYPE TypeName AS
(AttrName1 DataType1, AttrName2 DataType2, ...)

Example

```
Create type ComplexNumber as ( r float, i float );
```

```
Create type CourseInfo as (
     course UnswCourseCode,
     syllabus text,
     lecturer text
);
```

If attributes need constraints, can be supplied by using a DOMAIN.

User-defined Data Types(cont.)

CREATE TYPE is different from CREATE TABLE:

1. does not create a new (empty) table
2. does not provide for key constraints
3. does not have explicit specification of domain constraints

Used for **specifying return types of functions** that return tuples or sets.

PostgreSQL: SQL Functions

PostgreSQL allows users to define functions to be used in SQL

```
CREATE OR REPLACE FUNCTION
    funcName(arg1type, arg2type, ....)
    RETURNS rettype
AS $$
    SQL statements
$$ LANGUAGE sql;
```

PostgreSQL: SQL Functions(cont.)

Function arguments: accessed as $1, $2, ...

Return value: result of **the last** SQL statement.

- *rettype* can be any PostgreSQL data type.
- Rettype can be a table: *returns set of TupleType*

PostgreSQL: SQL Functions(cont.)

Example1:

```
-- max price of specified beer
create or replace function
      maxPrice(text) returns float
as $$
      select max(price) from Sells where beer = $1;
$$ language sql;
```

PostgreSQL: SQL Functions(cont.)

```
-- usage examples
select maxPrice('New’);
```

```
maxprice
----------
2.8
```

```
select bar, price from sells
where beer='New' and price=maxPrice('New’);
```

```
bar              price
---------------- -------
Marble Bar       2.8
```

PostgreSQL: SQL Functions(cont.)

Example2:

```
-- set of Bars from specified suburb
create or replace function
      hotelsIn(text) returns setof Bars
as $$
      select * from Bars where addr = $1;
$$ language sql;
```

PostgreSQL: SQL Functions(cont.)

```
-- usage examples
select * from hotelsIn('The Rocks');
```

```
name              addr            license
-------------     --------------- -----------
Australia Hotel   The Rocks       123456
Lord Nelson       The Rocks       123888
```

PL/pgSQL (PostgreSQL Manual: Chapter 43)

Procedural **L**anguage extensions to **P**ost**g**re**SQL**

A *PostgreSQL-specific language* integrating features of:

- procedural programming
- SQL programming

PL/pgSQL Function

PLpgSQL functions are created in the db :

```
CREATE OR REPLACE FUNCTION
    funcName(param1, param2, ....)
    RETURNS rettype
AS $$
    DECLARE
            variable declarations
    BEGIN
            code for function
    END;
$$ LANGUAGE plpgsql;
```

Note: the entire function body is a single SQL string.

PL/pgSQL Function Parameters

All parameters are passed by value in PL/pgSQL

Within a function, parameters can be referred:

- using positional notation ($1, $2, ...)

OR

- via aliases, supplied either
  - as part of the function header (e.g. f(a int, b int))
  - as part of the declarations (e.g. a alias for $1; b alias for $2)

PL/pgSQL Function Parameters(cont.)

Example: new-style function

```
CREATE OR REPLACE FUNCTION
       add(x text, y text) RETURNS
text
AS $$
DECLARE
       result text; -- local variable
BEGIN
       result := x||''''||y;
       return result;
END;
$$ LANGUAGE 'plpgsql';
```

Beware: never give aliases the same names as attributes.

PL/pgSQL Function Parameters(cont.)

Example: old-style function exists

```
CREATE OR REPLACE FUNCTION
      cat(text, text) RETURNS text
AS $$
DECLARE
      x alias for $1; -- alias for parameter
      y alias for $2; -- alias for parameter
      result text; -- local variable
BEGIN
      result := x||''''''''||y;
      return result;
END;
$$LANGUAGE ‘plpgsql’;
```

Beware: never give aliases the same names as attributes.

PL/pgSQL Function Parameters(cont.)

Restrictions: requires x and y to have values of the same "addable" type.

```
CREATE OR REPLACE FUNCTION
       add ( x any_element , y any_element ) RETURNS any_element
AS $$
BEGIN
       return x + y ;
END ;
$$ LANGUAGE plpgsql ;
```

PL/pgSQL Function Parameters(cont.)

PLpgSQL allows ***function overloading*** (i.e. same name, different arg types)

Example

```
CREATE FUNCTION add ( int , int ) RETURNS int AS
$$ BEGIN return $1 + $2 ; END ; $$ LANGUAGE plpgsql ;

CREATE FUNCTION add ( int , int , int ) RETURNS int AS
$$ BEGIN return $1 + $2 + $3 ; END ; $$ LANGUAGE plpgsql ;

CREATE FUNCTION add ( char (1) , int ) RETURNS int AS
$$ BEGIN return ascii ( $1 )+ $2 ; END ; $$ LANGUAGE plpgsql;
```

But must differ in arg types, so cannot also define:

```
CREATE FUNCTION add ( char (1) , int ) RETURNS char AS
$$ BEGIN return chr ( ascii ( $1 )+ $2 ); END ; $$ LANGUAGE plpgsql ;
```

i.e. cannot have two functions that look like add(char(1), int).

Function Return Types

A PostgreSQL function can return a value which is

- an atomic data type (e.g. integer, text, ...)
- a tuple (e.g. table record type or tuple type)
- a set of atomic values (like a table column)
- a set of tuples (i.e. a table)
- ***void*** (i.e. no return value)

A function returning a set of tuples is similar to a view.

Function Return Types(cont)

Examples of different function return types:

```
create type Employee as (id integer, name text, salary float, ...);
```

```
create function factorial(integer)
    returns integer ...
create function EmployeeOfMonth(date)
    returns Employee ...
create function allSalaries()
    returns setof float ...
create function OlderEmployees()
    returns setof Employee ...
```

Function Return Types(cont)

Different kinds of functions are invoked in different ways:

```
select factorial(5);
        -- returns one integer
select EmployeeOfMonth('2008-04-01');
        -- returns (x, y, z,...)
```

```
select * from EmployeeOfMonth('2008-04-01');
        -- one-row table
select * from allSalaries();
        -- single-column table
select * from OlderEmployees();
        -- subset of Employees
```

Using PL/pgSQL Functions

PLpgSQL functions can be invoked in several ways:

as part of a SELECT statement

```
select myFunction ( arg1 , arg2 );
select * from myTableFunction ( arg1 , arg2 );
```

as part of the execution of another PLpgSQL function

```
PERFORM myVoidFunction ( arg1 , arg2 );
result := myOtherFunction ( arg1 );
```

automatically via an insert/delete/update trigger

```
create trigger T before an update on R
for each row execute procedure myCheck ();
```

Declaring Data Types

Variables can also be defined in terms of:

- the type of an existing variable or table column
- the type of an existing table row (implicit RECORD type)

Declaring Data Types

The variable of a composite type is called a row-type variable.
A row-type variable can hold one row from a SELECT query result.

You can declare a variable to have the same type as a row from a table using <table_name>***%ROWTYPE***, e.g.

```
account Accounts%ROWTYPE ;
```

You may also refer to an attribute type using and specifying <table_name>***.*** <column_name>***%TYPE***, e.g.

```
account.branchName%TYPE
```

Declaring Data Types

Examples of declaring data types (in a PL/pgsql function)

- quantity INTEGER;
- start_quantity quantity%TYPE;
- employee Employees%ROWTYPE;
- name Employees.name%TYPE;

Control Structures in Pl/pgsql

Assignment

- variable := expression;

Example:

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

Conditionals

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSIF ... THEN ... ELSE

Example

```
IF v_user_id > 0 THEN
UPDATE users SET email = v_email WHERE user_id = v_user_id; END IF;
```

Control Structures(cont.)

Iteration

```
LOOP
    Statement
END LOOP;
```

Example

```
LOOP
    -- some computations
    EXIT WHEN count > 0;
END LOOP;
```

Control Structures(cont.)

Iteration

```
FOR int_var IN low .. high LOOP
    Statement
END LOOP ;
```

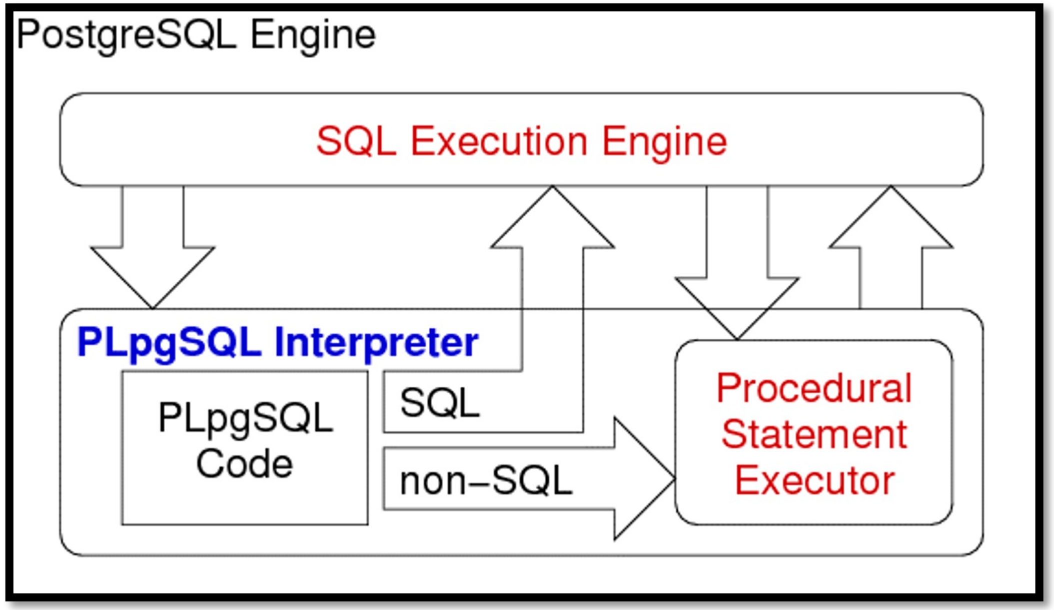
Example

```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;
```

PL/pgSQL(cont)

The PL/pgSQL interpreter

- executes procedural code and manages variables
- calls PostgreSQL engine to evaluate SQL statements



PL/pgSQL

Provided a means for extending DBMS functionality, e.g.

- implementing constraint checking (triggered functions)
- complex query evaluation (e.g. recursive)
- complex computation of column values
- detailed control of displayed results

PL/pgSQL Function

Stored-procedure approach (PLpgSQL):

```
create function
        withdraw(acctNum text, amount integer) returns text as $$
declare bal integer;
begin
        select balance into bal
        from Accounts
        where acctNo = acctNum;
        if (bal < amount) then
                return 'Insufficient Funds';
        else
                update Accounts
                set balance = balance - amount
                where acctNo = acctNum;
                select balance into bal
                from Accounts where acctNo = acctNum;
                return 'New Balance: ' || bal;
        end if;
end;
$$ language plpgsql;
```

SELECT ... INTO

Can capture query results via:

$$SELECT\ Exp_1, Exp_2, \ldots, Exp_n$$
$$INTO\ Var_1, Var_2, \ldots, Var_n$$
$$FROM\ TableList$$
$$\text{WHERE Condition} \ \ldots$$

The semantics:

1. Execute the query as usual
2. Return "projection list" ($Exp_1, Exp_2, \ldots$) as usual
3. Assign each Exp_i to corresponding Var_i

SELECT ... INTO (cont.)

Assigning a simple value via SELECT ... INTO:

```
-- cost is local var, price is attr
SELECT price INTO cost
FROM StockList
WHERE item = 'Cricket Bat';
cost := cost * (1 + tax_rate);
total := total + cost ;
```

Exceptions

Syntax of exceptions

```
BEGIN
    Statements ...
EXCEPTION
    WHEN Exceptions1 THEN
        StatementsForHandler1
    WHEN Exceptions2 THEN
        StatementsForHandler2
    ...
END;
```

Each Exception could be an OR list of exception names, e.g.,

- division_by_zero OR floating_point_exception OR ...

Exceptions(cont.)

Example:

```
-- Table T contains one tuple ('Tom', 'Jones')
DECLARE
    x INTEGER := 3;
BEGIN
    UPDATE T SET firstname = 'Joe' WHERE lastname = 'Jones';
    -- Table T now contains ('Joe', 'Jones')
    x := x + 1;
        y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
    -- update on T is rolled back to ('Tom', 'Jones')
    RAISE NOTICE 'Caught division_by_zero';
    RETURN x ;
    -- value returned is 4
END ;
```

Exceptions(cont.)

The ***RAISE*** operator generates server log entries, e.g.

- RAISE DEBUG ' Simple message ';
- RAISE NOTICE ' User = % ' , user_id ;
- RAISE EXCEPTION ' Fatal : value was % ' , value ;

There are several levels of severity:

- DEBUG, LOG, INFO, NOTICE, WARNING, and EXCEPTION
- not all severities generate a message to the client

Cursors

A cursor is an object that retrieves rows from a result table

A cursor is linked to a query, cursors move sequentially from row to row of a result table

Useful for applications to retrieve each row sequentially from the result table.

What happen when the cursor reaches the end of a result table?

Employees

Id	Name	Salary
961234	John Smith	35000.00
954321	Kevin Smith	48000.00
912222	David Smith	31000.00

cursor

Cursors

Benefits of cursors:

- Save network bandwidth and time. We don’t need to wait for whole result set to be retrieved/ processed.
- Since the cursor already stores the value of a row, other database processes can continue to update or delete other rows on the table,
- You can return a cursor in a pl/pgsql function.

Cursors(cont.)

A FOR loop works with a built-in cursor. There are also explicit cursors in pl/pgsql.
Requires: ***RECORD*** variable or ***Table%ROWTYPE*** variable

```
Create Function totalSalary() Returns real As $$
Declare
        employee RECORD;
        totalSalary REAL:=0;
Begin
        FOR employee IN SELECT * FROM Employees
        Loop
           totalSalary:=totalSalary+employee.salary ;
        End Loop;
        Return total;
End ; $$ Language plpgsql;
```

This style accounts for 95% of cursor usage.

Note:
the record type provided by PostgreSQL is like the row-type.
You *may* use only a single *row* in a record variable.

Opening and Closing Cursors

A cursor is usually bound to a specific query (i.e., a **bound cursor**)

<cursor_name_a> CURSOR FOR <query_b>;

OPEN <cursor_name_a>;

…

CLOSE <cursor_name_a>;

OR a cursor may be declared without reference to any query. A cursor that isn't bound to a query is an **unbound cursor**.

<cursor_name_c> REFCURSOR;

OPEN <cursor_name_c> FOR <query_d>; … CLOSE <cursor_name_c>;

OPEN <cursor_name_c> FOR <query_e>; …

Either way, declaring a cursor creates an **explicit** cursor.

Fetching Cursors

The fetch operator retrieves the next row from the cursor into a target.

```
FETCH e INTO me;
FETCH e INTO my_id , my_name , my_salary ;
```

Note: the variables need to match the corresponding type form the return table.

You could also use fetch in the opposite direction if you specified SCROLL in the cursor declaration.

E.g., `<cursor_name_a> SCROLL CURSOR FOR <query_b>;`

Cursors(cont.)

Example of operations on cursors:

```
DECLARE
   employee Employee%ROWTYPE;
   e CURSOR FOR Select * From Employees ;
         totalSalary REAL:=0;
Begin
   OPEN e;
   LOOP
         FETCH e INTO employee;
         EXIT WHEN NOT FOUND;
          totalSalary := totalSalary +employee.salary;
   END LOOP ;
   CLOSE e ;
End; …
```

Database Triggers(cont.)

The *event-condition-action* rules were developed to support the need to react to different kinds of *events* occurring in active databases

Most relational DBMSs effectively support ECA rules by using triggers or procedures, and triggers are included in the SQL:1999 standard.

Event-condition-action rules approach:

- an event activates the trigger
- on activation, the trigger checks a condition
- if the condition holds, a procedure is executed (the action)

In short: a set of stored procedures to automatically executed in response to specified database events

Database Triggers in PostgreSQL

Syntax for PostgreSQL trigger definition:

```
CREATE TRIGGER TriggerName
AFTER/BEFORE Event1 [OR Event2 ...]
ON TableName
FOR EACH ROW/STATEMENT
EXECUTE PROCEDURE FunctionName(args...);
```

Once a trigger is defined, it is bound to one or more database events.

PostgreSQL triggers provide a mechanism for INSERT, DELETE or UPDATE events to automatically activate PL/pgSQL functions

Trigger Procedures(cont.)

A trigger is defined, there needs to be a trigger procedure.

```
-- Create a trigger
```

```
CREATE TRIGGER TriggerName
…
EXECUTE PROCEDURE function_name(args...);
```

```
-- follow with the trigger procedure
```

```
CREATE OR REPLACE FUNCTION function_name() RETURNS
TRIGGER
...
```

Types of Triggers

Row level triggers and **Statement-level triggers**

- Row-level triggers executes once for each row affected in the transaction
- Statement-level trigger is invoked once per statement/transaction

CREATE TRIGGER TriggerName
AFTER/BEFORE Event1 ON TableName
*FOR EACH **ROW***
EXECUTE PROCEDURE FunctionName(args...);

CREATE TRIGGER TriggerName
AFTER/BEFORE Event1 ON TableName
*FOR EACH **STATEMENT***
EXECUTE PROCEDURE FunctionName(args...);

Trigger Procedures(cont.)

The trigger function also receives two variables **NEW** and **OLD** that contains the new and old row version, respectively.

Depending on the trigger, NEW and OLD variables can be accessed.

Trigger	NEW	OLD
Insert	Yes	No
Update	Yes	Yes
Delete	No	Yes

Possible usage: RETURN OLD or RETURN NEW (depending on which version of the tuple is to be used)

Trigger Example

Consider a database of people in the USA:

```
Create table Person (
   id integer primary key,
   ssn varchar(11) unique,
   state char(2), ... );
```

```
Create table States (
   id integer primary key,
   code char(2) unique,
   ... );
```

We want the state value
Person.state ∈ (select code from States), or
exists (select id from States where code=Person.state)

Note: we can use a trigger to help enforce this constraint.

Trigger Example(cont.)

Create Trigger checkState before insert or update on Person for each row execute procedure **checkState()**;

```
Create Function checkState() returns trigger as $$
begin
    -- normalize the user-supplied value
    new.state = upper(trim(new.state));
    if (new.state !~ '^[A-Z][A-Z]$') then
        raise exception 'Code Must Be Two Alpha Chars';
    end if;
    -- implement referential integrity check
    select * from States where code=new.state;
    if (not found) then
        raise exception 'Invalid State Code %',new.state;
    end if;
    return new;
end;  $$ language plpgsql;
```

Trigger Example(cont.)

Example Scenario:

- Employee(id, name, address, deptartment, salary)
- Department(id, name, manager, **totSal**)

Consider a **constraint** that we wish to enforce.

The value of Department.total_salary be equal to that of…

select sum(e.salary) from Employee e where e.dept = d.id;

Question: How can we keep the value of total_salary correct?

Trigger Example(cont.)

Example Scenario:

- Employee(id, name, address, deptartment, salary)
- Department(id, name, manager, **totSal**)

These natural events could affect the **validity of the database**

- a new employee beginning work in some department
- an employee getting a rise in salary
- an employee changing from one department to another
- an employee leaving the company

Trigger Example(cont.)

Case 1: A new employees arrives

```
Create trigger TotalSalary1
after insert on Employees
for each row execute procedure totalSalary1();
```

```
Create function totalSalary1() returns trigger
as $$
begin
        if (new.dept is not null) then
           update Department
            set totSal = totSal + new.salary where Department.id = new.dept;
        end if;
        return new;
end; $$ language plpgsql;
```

Trigger Example(cont.)

Case 2: An employees change departments/salaries

```
Create trigger TotalSalary2
after update on Employee
for each row execute procedure totalSalary2();
```

```
Create function totalSalary2() returns trigger
as $$
begin
    update Department
    set totSal = totSal + new.salary where Department.id = new.dept;
    update Department
    set totSal = totSal - old.salary  where Department.id = old.dept;
    return new;
end; $$ language plpgsql;
```

Trigger Example(cont.)

Case 3: An employee leaves

```
Create trigger TotalSalary3
after delete on Employee
for each row execute procedure totalSalary3();
```

```
Create function totalSalary3() returns trigger
as $$
begin
    if (old.department is not null) then
       update Department
       set totSal = totSal - old.salary where Department.id = old.deptartment;
    end if;
    return old;
end; $$ language plpgsql;
```

Database Triggers(cont.)

General database trigger usage scenarios:

1. To maintain a separate table for summary data
2. Checking schema-level **constraints** (assertions) on update
3. To perform updates across tables (to maintain assertions)

Trigger events(cont.)

Database triggers invoke automatically when the defined event occurs:

We've seen the following in action in the Trigger Example slides

- After Delete? Maintain summary data
- After Update? Maintain summary data
- After Insert? Maintain summary data

Think about situations where this is useful?

- Before Insert? Validate the format of the new data
- Before Delete?
- Before Update? Validate the format of the new data

All can be used to enforce constraints or business rules

UDF in PostgreSQL

A **User-Defined Function (UDF)** is a function that is written by the user and executed by the database.

PostgreSQL provides four kinds of functions:

- Query language functions (functions written in SQL)
- Procedural language functions (e.g., PL/pgSQL)
- Internal functions
- Functions in other programming language (e.g., C, Python, Java)