Extending Relational Databases

Limitations of Basic SQL

What we have seen of SQL so far:

- data definition language (**`create table(...)`**)
- constraints (domain, key, referential integrity)
- query language (**`select...from...where...`**)
- views (give names to SQL queries)

This is not sufficient to write complete applications.

More extensibility and programmability are needed.

Extending SQL

Ways in which standard SQL might be extended:

- new data types (incl. constraints, I/O, indexes, ...)
- object-orientation
- more powerful constraint checking

- packaging/parameterizing queries
- more functions/aggregates for use in queries
- event-based triggered actions

All are required to assist in application development.

∮ User-defined Data Types

SQL Data Types

SQL data definition language provides:

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

Other programming languages allow programmers add new types.

SQL also provides mechanisms to define new types:

- basic types: **CREATE DOMAIN ... AS** *BaseType* ...
- enumerated types: **CREATE TYPE ... AS ENUM (** *Vals* **)**
- tuple types: **CREATE TYPE ... AS (** *Attrs* **)**

SQL Data Types (cont)

Defining an atomic type (as specialisation 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, e.g.

```
create table Course (
    id     integer,
    code   UnswCourseCode,
    ...
);
```

SQL Data Types (cont)

Defining an enumerated type:

```
CREATE TYPE TypeName AS ENUM
( Str1, Str2, ... Strn )
```

Examples:

```
create type Color as enum
    ('red','orange','yellow','green','blue','violet');

create type Grade as enum
    ('FL','PS','CR','DN','HD');
```

Note: defines values and an ordering on the values

e.g. **'FL' < 'PS' < 'CR' < 'DN' < 'HD'**

SQL Data Types (cont)

Difference between domains and enumerated types:

```
create domain mood1 text
   check (value in ('sad','ok','happy'));

create type mood2 as enum ('sad','ok','happy');
```

Example:

```
select 'sad'::mood1 < 'happy'::mood1;
```

Returns false; values are compared as text strings.

```
select 'sad'::mood2 < 'happy'::mood2;
```

Returns true; values are compared as ordered enum constants.

SQL Data Types (cont)

Defining a tuple type:

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

Examples:

```
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**.

SQL Data Types (cont)

Other ways that tuple types are defined in SQL:

- **`CREATE TABLE`** *T* (effectively creates tuple type *T*)
- **`CREATE VIEW`** *V* (effectively creates tuple type *V*)

`CREATE TYPE` is different from **`CREATE TABLE`**:

- does **not** create a new (empty) table
- does **not** provide for key constraints
- does **not** have explicit specification of domain constraints

The main use of tuple types:
as result types of functions that return tuples or sets.

SQL Data Types (cont)

`CREATE TYPE` has one more very important use ...

For specifying full details of new (atomic) types:

- how to parse/display values of the type
- converting external to internal representation
- defining storage parameters for internal representation
- function for statistical analysis of type (for query optimiser)

See PostgreSQL Manual (e.g. SQL reference and Section 34.11) for details.

∮ Programming with SQL

SQL as a Programming Language

SQL is a powerful language for manipulating relational data.

But it is not a powerful programming language.

At some point in developing complete database applications

- we need to implement user interactions
- we need to control sequences of database operations
- we need to process query results in complex ways

and SQL cannot do any of these.

SQL cannot even do something as simple as factorial!

Ok ... so PostgreSQL added a factorial operator ... but it's non-standard.

What's wrong with SQL?

Consider the problem of withdrawal from a bank account:

If a bank customer attempts to withdraw more funds than they have in their account, then indicate "Insufficient Funds", otherwise update the account

An attempt to implement this in SQL:

```
select 'Insufficient Funds'
from   Accounts
where  acctNo = AcctNum and balance < Amount;
update Accounts
set    balance = balance - Amount
where  acctNo = AcctNum and balance >= Amount;
select 'New balance: '||balance
from   Accounts
where  acctNo = AcctNum;
```

What's wrong with SQL? (cont)

Two possible evaluation scenarios:

- displays "Insufficient Funds", **`UPDATE`** has no effect, displays unchanged balance
- **`UPDATE`** occurs as required, displays changed balance

Some problems:

- SQL doesn't allow parameterisation (e.g. *AcctNum*)
- always attempts **`UPDATE`**, even when it knows it's invalid

- need to evaluate **`(balance <`** *Amount***`)`** test twice
- always displays balance, even when not changed

To accurately express the "business logic", we need facilities like conditional execution and parameter passing.

Database Programming

Database programming requires a combination of

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

This combination is realised in a number of ways:

- passing SQL commands via a "call-level" interface
  (prog lang is decoupled from DBMS; most flexible; e.g. Java/JDBC, PHP)
- embedding SQL into augmented programming languages
  (requires pre-processor for language; typically DBMS-specific; e.g. SQL/C)

- special-purpose programming languages in the DBMS
  (closely integrated with DBMS; enable extensibility; e.g. PL/SQL, PLpgSQL)

Database Programming (cont)

Combining SQL and procedural code solves the "withdrawal" problem:

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

```
      end if
  end;
```

(This example is actually a stored procedure, using SQL/PSM syntax)

∮ Stored Procedures

Stored Procedures

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 database
- executed within the DBMS (close coupling with query engine)

Benefits of using stored procedures:

- code executed inside DBMS is fast with large data

- 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 = **P**ersistent **S**tored **M**odules)

Syntax for PSM procedure/function definitions:

```
CREATE PROCEDURE ProcName ( Params )
[ local declarations ]
procedure body ;

CREATE FUNCTION FuncName ( Params )
RETURNS Type
[ local declarations ]
function body ;
```

Parameters have three modes: **`IN`**, **`OUT`**, **`INOUT`**

SQL/PSM (cont)

SQL/PSM Syntax:

```
BEGIN statements END;

SET var = expression;
-- where expression may be an SQL query

IF cond_1 THEN statements_1
ELSIF cond_2 THEN statements_2
ELSE statements_n
END IF;
```

SQL/PSM (cont)

More SQL/PSM syntax:

```
LoopName: LOOP
    statements
    LEAVE LoopName;
    more statements
END LOOP;

WHILE condition DO
    statements
END WHILE;

FOR LoopName AS CursorName
    CURSOR FOR Query DO
    statements
END FOR;
```

SQL/PSM (cont)

Example: Find cost of Toohey's New beer at a specified bar

Default behaviour: return price charged for New at that bar.

```
function CostOfNew(string) returns float;
```

How to deal with the case: New is not sold at that bar?

- use exception-handling (e.g. Java)
- return null or negative value to indicate error
- return two values: price and/or status

In PSM, could use return-value plus **`OUT`**-mode parameter.

SQL/PSM (cont)

Using a **`NULL`** return value ...

```
CREATE FUNCTION
    CostOfNew(IN pub VARCHAR)
    RETURNS FLOAT
DECLARE cost FLOAT;
BEGIN
    SET cost = (SELECT price FROM Sells
                WHERE  beer = 'New' and
                       bar = pub);
    -- cost is null if not sold in bar
    RETURN cost;
END;
```

SQL/PSM (cont)

How this function is used:

```
DECLARE myCost FLOAT;
...
SET myCost = CostOfNew('The Regent',ok);
IF (myCost is not null) THEN
    ... do something with the cost ...
ELSE
    ... handle not having a cost ...
END IF;
```

SQL/PSM (cont)

Using an **`OUT`** parameter ...

```
CREATE FUNCTION
    CostOfNew(IN pub VARCHAR,
              OUT status BOOLEAN)
    RETURNS FLOAT
DECLARE cost FLOAT;
BEGIN
    SET cost = (SELECT price FROM Sells
                WHERE  beer = 'New' and
                       bar = pub);
    SET status = (cost IS NOT NULL);
    RETURN cost;
END;
```

SQL/PSM (cont)

How this function is used:

```
DECLARE myCost FLOAT;
DECLARE ok BOOLEAN;
...
SET myCost = CostOfNew('The Regent',ok);
IF (ok) THEN
```

```
    ... do something with the cost ...
ELSE
    ... handle not having a cost ...
END IF;
```

SQL/PSM (cont)

Example: Find the sum of the first 100 integers.

Without using any functions at all:

```
DECLARE i integer;
DECLARE sum integer;
...
SET sum = 0; SET i = 1;
WHILE (i <= 100) DO
    SET sum = sum + i;
    SET i = i + 1;
END WHILE;
```

SQL/PSM (cont)

Sum(100) using a regular function to add two numbers:

```
CREATE FUNCTION
    add(IN a integer, IN b integer) RETURNS integer
BEGIN
    return a + b;
END;
```

which would be used as:

```
WHILE (i < 20) DO
    set sum = add(sum,i);
    set i = i + 1;
END WHILE;
```

SQL/PSM (cont)

Sum(100) using a procedure with an **`INOUT`** parameter:

```
CREATE PROCEDURE
    accum(INOUT sum integer, IN val integer)
```

```
BEGIN
    set sum = sum + val;
END;
```

which would be used as:

```
WHILE (i < 20) DO
    accum(sum,i);
    set i = i + 1;
END WHILE;
```

PSM in Real DBMSs

Unfortunately, the PSM standard was developed after most DBMSs had their own stored procedure language

⇒ no DBMS implements the PSM standard exactly.

IBM's DB2 and MySQL implement the SQL/PSM closely (but not exactly)

Oracle's PL/SQL is moderately close to the SQL/PSM standard

- syntax differences e.g. **`EXIT`** vs **`LEAVE`**, **`DECLARE`** only needed once, ...
- extra programming features e.g. packages, exceptions, input/output

PostgreSQL's PLpgSQL is close to PL/SQL (95% compatible)

- has only functions (but can return **`void`**); limited exceptions; no i/o
- PLpgSQL function bodies are defined within a string
- PLpgSQL is just one of a number of languages for stored procedures

---

∮ PLpgSQL

(PostgreSQL Manual: Chapter 38: PLpgSQL)

PLpgSQL

PLpgSQL = **P**rocedural **L**anguage extensions to **P**ost**g**re**SQL**

A PostgreSQL-specific language integrating features of:

- procedural programming and SQL programming

Functions are stored in the database with the data.

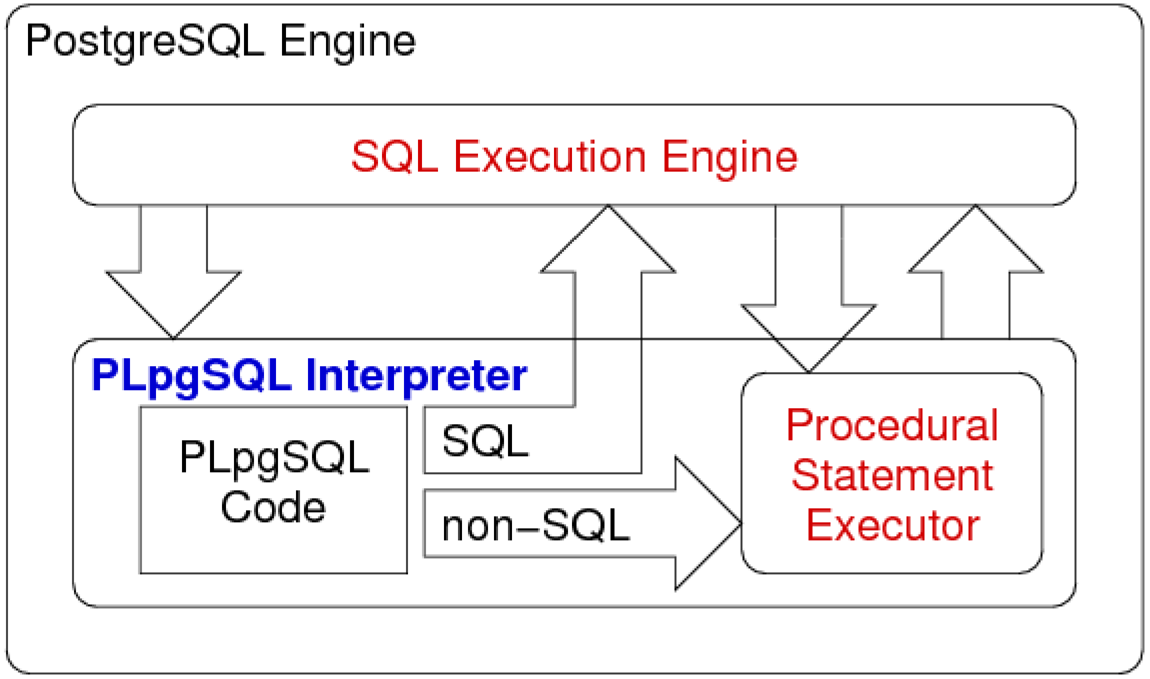
Provides a means for extending DBMS functionality, e.g.

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

PLpgSQL (cont)

The PLpgSQL interpreter

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



Defining PLpgSQL Functions

PLpgSQL functions are created (and inserted into db) via:

```
CREATE OR REPLACE
    funcName(arg1type, arg2type, ....)
    RETURNS rettype
AS '
DECLARE
    variable declarations
```

```
BEGIN
   code for function
END;' LANGUAGE plpgsql;
```

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

Defining PLpgSQL Functions (cont)

Reasons for defining functions as strings:

- allows functions to be defined in different languages
- simplifies the SQL parser

Problems with defining functions as strings:

- requires a change of "lexical level"
- leads to complex, error-prone definitions like:

```
create function Fun(name text) returns text
```

```
as '
begin  return ''It''''s fun, ''||name||''!'';  end;
' language plpgsql;
```

To fix the last problem, PostgreSQL 8 has introduced a new quoting mechanism, somewhat like "here-documents" in shell/Perl/PHP.

Defining PLpgSQL Functions (cont)

Comparison of old/new quoting mechanisms for functions:

```
-- old style quoting
create function Fun(name text) returns text
as '
begin
    return ''It''''s fun, ''||name||''!'';
end;
' language plpgsql;

-- new style quoting
create function Fun(name text) returns text
as $$
begin
```

```
    return 'It''s fun, '||name||'!';
end;
$$ language plpgsql;
```

The **$$** may contain an embedded identifier (e.g. the function name).

Defining PLpgSQL Functions (cont)

Solution to "withdrawal" problem in PLpgSQL:

```
CREATE OR REPLACE FUNCTION
   withdraw(acctNum text, amount real) RETURNS text AS $$
DECLARE
   current REAL;  newbalance REAL;
BEGIN
   SELECT INTO current balance
   FROM Accounts WHERE  acctNo = acctNum;
   IF (amount > current) THEN
      return 'Insufficient Funds';
   ELSE
      newbalance := current - amount;
      UPDATE Accounts
```

```
            SET    balance = newbalance
            WHERE  acctNo = acctNum;
            return 'New Balance: '||newbalance;
         END IF;
      END;
      $$ LANGUAGE plpgsql;
```

Defining PLpgSQL Functions (cont)

If a PLpgSQL function definition is syntactically correct

- the function is stored in the database
- but is not completely checked until executed

Common errors:

- using a variable with same name as some attribute

  (the variable hides the attribute, so queries using the attribute fail "inexplicably")
- forgetting to use `''` in body if using old-style quoting

Warning: PLpgSQL's error messages can sometimes be obscure.

However, the PLpgSQL parser and error messages have improved considerably in recent versions.

PLpgSQL Function Parameters

All parameters are passed by value in PLpgSQL.

Within a function, parameters can be referred to:

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

Nowadays, the last way is preferred (i.e. like "normal" functions)

PLpgSQL Function Parameters (cont)

Example: a function to add two integers (old style)

```
CREATE OR REPLACE FUNCTION
    add(int, int) RETURNS int
AS '
DECLARE
    x alias for $1;    -- alias for parameter
    y alias for $2;    -- alias for parameter
    sum integer;       -- local variable
BEGIN
    sum := x + y;
    return sum;        -- return result
END;
' LANGUAGE plpgsql;
```

Beware: do not ever give parameters the same names as attributes.

PLpgSQL Function Parameters (cont)

Example: a function to add two integers (new style)

```
CREATE OR REPLACE FUNCTION
    add(x int, y int) RETURNS int
AS $add$
DECLARE
    sum integer;      -- local variable
BEGIN
    sum := x + y;
    return sum;       -- return result
END;
$add$ LANGUAGE plpgsql;
```

Beware: do not ever give parameters the same names as attributes.

PLpgSQL Function Parameters (cont)

Example: a function to add two values (polymorphic)

```
CREATE OR REPLACE FUNCTION
    add(x anyelement, y anyelement) RETURNS anyelement
AS $add$
BEGIN
    return x + y;
```

```
END;
$add$ LANGUAGE plpgsql;
```

Restrictions:

- requires **`x`** and **`y`** to have values of the same "add-able" type
- does not allow definition of **`sum anyelement`** to hold result

PLpgSQL Function Parameters (cont)

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

Examples:

```
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 result 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, float, boolean, ...`**)
- 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)

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

Function Return Types (cont)

Examples of different function return types:

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

The **`OlderEmployees`** function returns an instance of the **`Employee`** table.

Functions can also return "generic" tuples:

```
create function f(int) returns setof record ...
-- which is used as e.g.
select * from f(3) as T(a integer, b float) where ...
```

The last example says that "the call **`f(3)`** returns a table T(a,b)".

Function Return Types (cont)

Different kinds of functions are invoked in different ways:

- function fD() returning a single atomic data value

```
select fD();   -- like an attribute called fD
```

- function fT() returning a single tuple (record)

```
select fT();   -- like a (x,y,z) tuple-value
select * from fT() ... -- like a 1-row table
```

- function fS() returning set of atomic values or records

```
select * from fS() ... -- like a table called fS
```

Using PLpgSQL Functions

PLpgSQL functions can be invoked in several contexts:

- as part of a **SELECT** statement

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

(either on the command line or within another PLpgSQL function)

- 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 update on R
  for each row execute procedure myCheck();
```

What's Wrong with PLpgSQL?

Some things to beware of:

- doesn't provide any i/o facilities (except **`RAISE NOTICE`**)
  - the aim is to build computations on tables that SQL alone can't do
- functions are not syntax-checked when loaded into DB
- error messages are sometimes not particularly helpful

Summary: debugging PLpgSQL can sometimes be tricky.

What's Wrong with PLpgSQL? (cont)

Some other "deficiencies", compared to Oracle's PL/SQL

- no fine-grained transaction control within functions
  - deliberate design decision to simplify PostgreSQL functions
  - each function executes as part of a single transaction
- functions are defined as strings
  - change of "lexical scope" can sometimes be confusing
- (slightly) less powerful exception handling mechanism
  - can **`RAISE`** exception to abort function/transaction
  - can also use **`RAISE`** to report problems and continue

Nowadays, PLpgSQL has almost the same syntax as PL/SQL.

Data Types

PLpgSQL constants and variables can be defined using:

- standard SQL data types (**`CHAR, DATE, NUMBER, ...`**)
- user-defined PostgreSQL data types (e.g. **`Point`**)
- a special structured record type (**`RECORD`**)
- table-row types (e.g. **`Branches%ROWTYPE`**)
- types of existing variables (e.g. **`Branches.location%TYPE`**)

There is also a **`CURSOR`** type for interacting with SQL.

Data Types (cont)

Record variables are defined:

- using a "placeholder" **`RECORD`** type, e.g.

```
account RECORD;
```

(the actual type is fixed when the variable is bound to a query)

- by deriving a type from an existing database table, e.g.

```
account Accounts%ROWTYPE;
```

- as aliases for "tuple type" parameters, e.g.

```
CREATE FUNCTION summary(Accounts)
RETURNS integer AS '
DECLARE account alias for $1 ...
```

Record components referenced via attribute name   e.g. **`account.branchName`**

Data Types (cont)

Variables can also be defined in terms of:

- the type of an existing variable or table column
- the type of an existing table row (implict **`RECORD`** type)

Examples:

```
quantity     INTEGER;
start_qty    quantity%TYPE;

employee     Employees%ROWTYPE;

name         Employees.name%TYPE;
```

Syntax/Control Structures

A standard assignment operator is available:

Assignment	*var* **:=** *expr* **SELECT** *expr* **INTO** *var*

Selection	`IF` C_1 `THEN` S_1
	 `ELSIF` C_2 `THEN` S_2 `...`
	 `ELSE` S `END IF`

Iteration	`LOOP` S `END LOOP`
	 `WHILE` C `LOOP` S `END LOOP`

```
FOR rec_var IN Query LOOP ...
FOR int_var IN lo..hi LOOP ...
```

SELECT...INTO

Can capture query results via:

```
SELECT  Exp1,Exp2,...,Expn
INTO    Var1,Var2,...,Varn
FROM    TableList
WHERE   Condition ...
```

The semantics:

- execute the query as usual
- return "projection list" (Exp_1,Exp_2,...) as usual
- 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;
```

The current PostgreSQL parser also allows this syntax:

```
SELECT INTO cost price
FROM   StockList
WHERE  item = 'Cricket Bat';
```

SELECT...INTO (cont)

Assigning whole rows via **SELECT...INTO**:

```
DECLARE
   emp     Employees%ROWTYPE;
   eName   text;
   pay     real;
BEGIN
   SELECT * INTO emp
   FROM Employees WHERE id = 966543;
   eName := emp.name;
   ...
   SELECT name,salary INTO eName,pay
   FROM Employees WHERE id = 966543;
END;
```

SELECT...INTO (cont)

In the case of a PLpgSQL statement like

```
select a into b from R where ...
```

If the selection returns no tuples

- the variable **b** gets the value **NULL**

If the selection returns multiple tuples

- the variable **b** gets the value from the first tuple

SELECT...INTO (cont)

If the above behaviour is too "generous", try:

```
select a into strict b from R where ...
```

If the selection returns no tuples

- the exception **NO_DATA_FOUND** is thrown

If the selection returns multiple tuples

- the exception **TOO_MANY_ROWS** is thrown

This behaviour matches Oracle's default behaviour.

SELECT...INTO (cont)

An alternative way of tracking **NO_DATA_FOUND** ...

Use the special variable **FOUND** ...

- local to each function, set false at start of function
- set true if a **SELECT** finds at least one tuple
- set true if **INSERT/DELETE/UPDATE** affects at least one tuple
- otherwise, remains as **FALSE**

Example of use:

```
select a into b from R where ...
if (not found) then
    -- handle case where no matching tuples b
```

INSERT ... RETURNING

Can capture values from tuples inserted into DB:

```
INSERT INTO Table(...) VALUES
(Val_1, Val_2, ... Val_n)
RETURNING ProjectionList INTO VarList
```

Useful for recording id values generated for **serial** PKs:

```
declare newid integer;
...
insert into T(id,a,b,c) values (default,2,3,'red')
returning id into newid;
-- which used to be done as ...
select nextval('T_id_seq') into newid;
insert into T(id,a,b,c) values(newid,2,3,'red');
```

Exceptions

PostgreSQL 8 introduced execption handling to PLpgSQL:

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

Each *Exceptions$_i$* is an **OR** list of exception names, e.g.

```
division_by_zero OR  floating_point_exception OR ...
```

A list of exceptions is in Appendix A of the PostgreSQL Manual.

Exceptions (cont)

When an exception occurs:

- control is transferred to the relevant exception handling code

- all database changes so far in this transaction are undone
- all function variables retain their current values
- handler executes and then transaction aborts (and function exits)

If no handler at given scoping level, exception passed to next outer level.

Default exception handlers at outermost level simply exit and log error.

Exceptions (cont)

Example of exception handling:

```
-- 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

`RAISE EXCEPTION` also generates a **`RAISE_EXCEPTION`** exception.

The server log for your PostgreSQL server is located in /srvr/*YOU*/pgsql823/logfile

---

Cursors

A cursor is a variable that can be used to access the result of a particular SQL query.

[Diagram:Pic/cursor.png]

Cursors move sequentially from row to row (cf. file pointers in C).

Cursors (cont)

Simplest way to use cursors: implicitly via **`FOR...IN`**

Requires: **`RECORD`** variable or *Table*`%`**`ROWTYPE`** variable

Example:

```
CREATE FUNCTION totsal() RETURNS REAL AS $$
DECLARE
   emp RECORD;   total REAL := 0;
BEGIN
   FOR emp IN SELECT * FROM Employees
   LOOP
      total := total + emp.salary;
   END LOOP;
   return total;
END; $$ LANGUAGE plpgsql;
```

This style accounts for 95% of cursor usage.

Cursors (cont)

Sidetrack ...

Of course, the previous example would be better done as:

```
CREATE FUNCTION totsal() RETURNS REAL AS $$
DECLARE
```

```
    total REAL;
BEGIN
    SELECT sum(salary) INTO total FROM Employees;
    return total;
END;$$ LANGUAGE plpgsql;
```

The iteration/summation can be done much more efficiently as an aggregation.

Cursors (cont)

Sidetrack ... (cont.)

It could also be done as a view:

```
CREATE VIEW totsalView AS
SELECT sum(salary) INTO total FROM Employees;
```

But note the different usage:

```
-- Which departments have a budget
-- greater than the total salary bill?
SELECT d.name FROM Department
```

```
WHERE d.budget > totsal();
-- versus
SELECT d.name FROM Department
WHERE d.budget > (SELECT * FROM totsalView);
```

Cursors (cont)

Basic operations on cursors: **OPEN**, **FETCH**, **CLOSE**:

```
-- assume ... e CURSOR FOR SELECT * FROM Employees;
OPEN e;
LOOP
   FETCH e INTO emp;
   EXIT WHEN NOT FOUND;
   total := total + emp.salary;
END LOOP;
CLOSE e;
...
```

The **FETCH** operation can also extract components of a row:

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

There must be one variable, of the correct type, for each column in the result.

Note: low-level cursor operations are rarely used in practice.

Cursors (cont)

Ways to declare cursors:

```
DECLARE
    a REFCURSOR;    -- unbound cursor
    b CURSOR FOR    -- bound cursor
      SELECT * FROM Emp WHERE salary > $1;
    -- parameterised cursor
    c CURSOR (base real) IS
      SELECT * FROM Emp WHERE salary > base;
BEGIN
    -- all access the same result set
    OPEN a FOR SELECT * FROM Emp WHERE salary > $1;
    OPEN b;
    OPEN c($1);
    ...
END;
```

Dynamically Generated Queries

EXECUTE takes a string and executes it as an SQL query.

Examples:

```
EXECUTE 'SELECT * FROM Employees';
EXECUTE 'SELECT * FROM '||'Employees';
EXECUTE 'SELECT * FROM '||quote_ident($1);
EXECUTE 'DELETE FROM Accounts '||
        'WHERE holder='||quote_literal($1);
```

EXECUTE *string* can be used in any context where the query *string* could have been used.

This mechanism allows us to construct queries "on the fly".

Dynamically Generated Queries (cont)

Example: a wrapper for updating a single text field

```
CREATE OR REPLACE FUNCTION set(TEXT,TEXT,TEXT) RETURNS INT
AS $$
DECLARE
   theTable alias for $1;  theField alias for $2;
   theValue alias for $3;  query TEXT;
BEGIN
   query := 'UPDATE ' || quote_ident(theTable);
   query := query || ' SET ' || quote_ident(theField);
   query := query || ' = ' || quote_literal(theValue);
   EXECUTE query;
   RETURN NULL;
END; $$ LANGUAGE plpgsql;
```

which could be used as e.g.

```
SELECT set('branches','address','Beach St.');
```

Dynamically Generated Queries (cont)

One limitation of **`EXECUTE`**:

- cannot use **`SELECT INTO`** in dynamic queries

Needs to be expressed instead as:

```
tuple R%rowtype; n int;
EXECUTE 'select * from R where id='||n INTO tuple;
-- or
x int; y int; z text;
EXECUTE 'select a,b,c from R where id='||n INTO x,y,z;
```

Notes:

- if query returns multiple tuples, first one is stored
- if query returns zero tuples, all nulls are stored

Functions Returning Tables

PLpgSQL functions can return tables by using a return type

```
CREATE OR REPLACE
    funcName(arg1type, arg2type, ....)
    RETURNS SETOF rowType
```

Example:

```
CREATE OR REPLACE FUNCTION
   valuableEmployees(REAL) RETURNS SETOF Employees
AS $$
DECLARE
   e RECORD;
BEGIN
   FOR e IN SELECT * FROM Employees WHERE salary > $1
   LOOP
      RETURN NEXT e;  -- accumulates tuples
   END LOOP;
   RETURN;  -- returns accumulated tuples
END; $$ language plpgsql;
```

Functions Returning Tables (cont)

Functions returning **SETOF** *rowType* are used like tables.

Example:

```
select * from valuableEmployees(50000);
 id |   name   | salary
```

```
----+--------+--------
   1 | David  |   75000
   2 | John   |   70000
   3 | Andrew |   75000
   4 | Peter  |   55000
   8 | Wendy  |   60000
(5 rows)
```

`SETOF` functions look similar to parameterised views.

Functions Returning Tables (cont)

A difference between views and functions returning a **`SETOF`**:

- **`CREATE VIEW`** produces a "virtual" table definition
  (table definitions induce a row type with same name as table e.g. **`Accounts`**)
- **`SETOF`** functions require an existing tuple type

In examples above, we used existing **`Employees`** tuple type.

In general, you need to define the tuple return type via

```
CREATE TYPE NewTupleType AS (
     attr_1  type_1,
     attr_2  type_2,
     ...
     attr_n  type_n
);
```

Functions Returning Tables (cont)

Example of using tuple types ... valuableEmployees() revisited:

```
CREATE TYPE EmpInfo as
   name  varchar(50),
   pay   integer
);
CREATE OR REPLACE FUNCTION
   valuableEmployees(REAL) RETURNS SETOF EmpInfo
AS $$
DECLARE
   emp RECORD;
   inf EmpInfo%ROWTYPE;
BEGIN
   FOR emp IN SELECT * FROM Employees WHERE salary > $1
```

```
      LOOP
         inf.name := emp.name;  inf.pay = emp.salary;
         RETURN NEXT inf;  -- accumulates tuples
      END LOOP;
      RETURN;  -- returns accumulated tuples
   END; $$ LANGUAGE plpgsql;
```

SQL Functions

PostgreSQL functions require you to specify a language.

In our examples, we have used primarily PLpgSQL.

Other PostgreSQL function languages: SQL, Tcl, Perl, ...

SQL functions provide a mechanism for parameterised views.

SQL Functions (cont)

Recall the **`ValuableEmployees`** example from above.

If we know that the minimum salary for a valuable employee will always be $50,000, we can solve the problem very simply as:

```
create or replace view ValuableEmployees as
select * from Employees where salary > 50000;
```

SQL Functions (cont)

If we want to allow minimum valuable salary to change, we need a way of replacing $50,000 by a supplied value.

SQL functions provide a simple mechanism for this:

```
create or replace function
    ValuableEmployees(integer) returns setof Employees
as $$
select * from Employees where salary > $1
$$ language sql;
```

SQL Functions (cont)

Differences between SQL and PLpSQL functions

- SQL function bodies are a single SQL statement
- SQL functions cannot use named parameters

  (required to use positional parameter notation: $1, $2, $3)
- SQL functions have no **`RETURN`**

  (their result is the result of the SQL statement)
- return types can be atomic, tuple, or **`setof`** tuples

SQL Functions (cont)

Comparison of SQL and PLpgSQL functions:

```
create function add(int,int) returns int
as $$ begin return ($1 + $2); end;
$$ language plpgsql;

create function add(int,int) returns int
as $$ select $1 + $2 $$ language sql;
```

```
create function fac(n int) returns int
as $$
begin
    if (n = 0) then return 1;
    else return n * fac(n-1);
    end if;
end;
$$ language plpgsql;

create function fac(int) returns int
as $$
   select case when $1 = 0 then 1
          else $1 * fac($1-1) end
$$ language sql;
```

SQL Functions (cont)

More comparison of SQL and PLpgSQL functions:

```
create or replace function
   valuableEmployees(REAL) returns setof Employees
as $$
    select * from Employees where salary > $1
$$ language sql;

create or replace function
```

```
   valuableEmployees(REAL) returns setof Employees
as $$
declare
   e record;
begin
   for e in select * from Employees where salary > $1
   loop  return next e;  end loop;
   return;
end; $$ language plpgsql;
```

Extra Thoughts on Functions

PostgreSQL provides a variety of abstraction mechanisms.

Always try to define "functions" as simply as possible.

E.g.

- best defined as view? SQL function? PLpgSQL function?
- if PLpgSQL, do we need explicit cursor or is **`FOR`** loop ok?
- do we need to dynamically construct a query string?
- or can we simply use substitution of parameter values?

Further Examples

More examples of PLpgSQL procedures may be found in

- the PostgreSQL documentation

  /home/cs3311/web/08s1/doc/pgsql830/plpgsql.html

- the PostgreSQL distribution

  /home/jas/systems/postgresql-8.2.3/src/test/regress/sql/plpgsql.sql

- the OpenACS system (web content management)

  /home/jas/systems/openacs-3.2.5/www/doc/sql (OpenACS makes extensive use of stored procedures and triggers)

∮ User-defined Aggregates

Aggregates

Aggregates reduce a collection of values into a single result.

Examples: **count(***Tuples***)**, **sum(***Numbers***)**, **avg(***Numbers***)**, etc.

The action of an aggregate function can be viewed as:

```
AggState = initial state
for each item V {
    # incorporate V into AggState
    AggState = newState(AggState, V)
}
return final(AggState)
```

Aggregates (cont)

Defining a new aggregate in PostgreSQL requires:

- description of the input (base) item type

- description of the state type
- value(s) for the initial state
- state transition function
- function to compute result from final state (optional)

Aggregates (cont)

New aggregates defined using **`CREATE AGGREGATE`** statement:

```
CREATE AGGREGATE AggName (
    basetype  = BaseType,
    stype     = StateType,
    initcond  = InitialValue,
    sfunc     = NewStateFunction,
    finalfunc = FinalResFunction
);
```

`initcond` is optional; defaults to **`NULL`**

`finalfunc` is optional; defaults to identity function

Aggregates (cont)

The state transition function always has type:

```
function newState(StateType,BaseType) returns StateType
```

The final function always has type

```
function finalValue(StateType) returns ResultType
```

ResultType may be the same as the *StateType*
or may be a component of the *StateType*

Aggregates (cont)

Example: **`sum2`** sums two columns of integers

```
create type IntPair as (x int, y int);

create function
    AddPair(sum int, p IntPair) returns int
```

```
as $$
begin return p.x+p.y+sum; end;
$$ language plpgsql;

create aggregate sum2 (
    basetype  = IntPair,
    stype     = int,
    initcond  = 0,
    sfunc     = AddPair
);
```

Constraints and Assertions

Constraints

So far, we have considered several kinds of constraints:

- attribute (column) constraints
- tuple (row) constraints
- relation (table) constraints
- referential integrity constraints

Examples:

```
create table Employee (
   id      integer primary key,
   name    varchar(40),
   salary  real,
   age     integer check (age > 15),
   worksIn integer
              references Department(id),
   constraint PayOk check (salary > age*1000)
);
```

An Aside on Constraints

When discussing SQL DDL, we indicated that attribute constraints could not involve queries on other tables.

E.g. it is not possible to specify something like:

```
create table R (x integer, y integer);
create table S (
    a integer check (a > (select max(x) from R)),
    b integer check (b not in (select y from R))
);
```

An Aside on Constraints (cont)

In fact, it is possible to implement arbitrary constraints

- define a function returning boolean
- pass the attribute(s) to be checked as parameters
- perform arbitrary checks in the function

Example:

```
create table R (x integer, y integer);
create table S (
    a integer check (biggerThanX(a)),
    b integer check (notInY(b))
);
```

An Aside on Constraints (cont)

Where **biggerThanX** and **notInY** are defined as:

```
create function biggerThanX(a integer) returns boolean
as $$
declare mx integer;
begin
    select max(x) into mx from R;
    return (a > mx);
end;
$$ language plpgsql;

create function notInY(b integer) returns boolean
as $$
begin
    select * from R where y = b;
    return (not FOUND);
end;
$$ language plpgsql;
```

Assertions

Column and table constraints ensure validity of one table.

RI constraints ensure connections between tables are valid.

In order to specify the conditions for validity of an entire database, we need to to be able to express more complex multi-table constraints.

Simple example:

```
for all Branches b
    b.assets = (select sum(acct.balance)
                from   Accounts acct
                where  acct.branch = b.location)
```

i.e. the assets of a branch is the sum of balances of accounts held at that branch

---

Assertions (cont)

Assertions are schema-level constraints

- typically involving multiple tables
- expressing a condition that must hold at all times

- need to be checked for each update on relevant tables
- cause update to be rejected if check fails

Usage:

```
CREATE ASSERTION name CHECK (condition)
```

Assertions (cont)

Example: no course at UNSW is allowed more than 999 enrolments

```
create assertion ClassSizeConstraint check (
    not exists (
        select c.id from Course c, Enrolment e
        where  c.id = e.course
        group  by c.id
        having count(e.student) > 999
    )
)
```

Needs to be checked

- after each change to **`Enrolment`** (changes to **`Course`**?)
- however the change occurs (e.g explicit insert,cascaded delete,...)

Assertions (cont)

Example: the assets of a bank branch are the sum of its account balances

```
create assertion AssetsCheck check (
    not exists (
        select branchName from Branch b
        where  b.assets <>
               (select sum(a.balance) from Accounts a
                         where a.branch = b.location)
    )
)
```

Needs to be checked

- after each change to **`Branch`** or **`Account`**

In this example, it might be more useful if we could *force* this condition to hold after account updates.

Assertions (cont)

On each update, it is expensive

- to determine which assertions need to be checked
- to run the queries which check the assertions

A database with many assertions would be **very** slow.

So, most RDBMSs do not implement general assertions.

Triggers are provided as

- a lightweight mechanism for dealing with assertions
- a general event-based programming tool for databases

Triggers

Triggers

Triggers are

- procedures stored in the database
- activated in response to database events (e.g.updates)

Active databases = databases using triggers extensively.

Examples of uses for triggers:

- checking schema-level constraints on update
- maintaining summary data
- performing multi-table updates (to maintain assertions)

Triggers (cont)

Triggers provide event-condition-action (ECA) programming:

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

Some typical variations on this:

- execute the action before, after or instead of the triggering event
- can refer to both old and new values of updated tuples
- can limit updates to a particular set of attributes
- perform action: once for each modified tuple, once for all modified tuples

---

Triggers (cont)

SQL "standard" syntax for defining triggers:

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE}  Event1 [ OR Event2 ... ]
[ FOR EACH ROW ]
ON TableName
```

```
[ WHEN ( Condition ) ]
Block of Procedural/SQL Code ;
```

Possible *Events* are **INSERT**, **DELETE**, **UPDATE**.

FOR EACH ROW clause ...

- if present, code is executed on each modified tuple
- if not present, code is executed once after all tuples are modified, just before changes are finally **COMMIT**ed

Example Trigger

Example: department salary totals

Scenario:

```
Employee(id, name, address, dept, salary, ...)
Department(id, name, manager, totSal, ...)
```

An assertion that we wish to maintain:

```
create assertion TotalSalary check (
    not exists (
        select d.id from Department d
        where  d.totSal <>
               (select sum(e.salary) from Employee e
                                 where e.dept = d.id)
    )
)
```

Example Trigger (cont)

Events that might affect the validity of the database

- a new employee starts work in some department
- an employee gets a rise in salary
- an employee changes from one department to another
- an employee leaves the company

A single assertion could check for this after each change.

With triggers, we have to program each case separately.

Each program implements updates to *ensure* assertion holds.

Example Trigger (cont)

Case 1: new employees arrive

```
create trigger TotalSalary1
after insert on Employee
for each row when (NEW.dept is not null)
   update Department
   set totSal = totSal + NEW.salary
   where  Department.id = NEW.dept;
```

Case 2: employees get a pay rise

```
create trigger TotalSalary2
after update of salary on Employee
for each row when (NEW.dept is not null)
   update Department
   set totSal = totSal + NEW.salary - OLD.salary
   where  Department.id = NEW.dept;
```

Example Trigger (cont)

Case 3: employees change departments

```
create trigger TotalSalary3
after update of dept on Employee
for each row
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;
```

Case 4: employees leave

```
create trigger TotalSalary4
after delete on Employee
for each row when (OLD.dept is not null)
   update Department
   set totSal = totSal - OLD.salary
   where  Department.id = OLD.dept;
```

Triggers in PostgreSQL

PostgreSQL triggers provide a mechanism for

- **INSERT**, **DELETE** or **UPDATE** events
- to automatically activate PLpgSQL functions

Syntax for PostgreSQL trigger definition:

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

Triggers in PostgreSQL (cont)

Examples of PostgreSQL trigger definitions:

```
-- check for each new Employee
create trigger checkEmpInsert
before insert on Employees
for each row
execute procedure checkInputValues();

create function checkInputValues() ...

-- check after all Employees changed
create trigger afterEmpChange
after update on Employees
for each statement
execute procedure fixOtherTables();

create function fixOtherTables() ...
```

Triggers in PostgreSQL (cont)

PostgreSQL does not have conditional activation of triggers (i.e. no **`WHEN`** clause in the trigger definition statement).

However, tests in the function can effectively provide this, e.g.

```
create trigger X before insert on T
when (C) begin ProcCode end;
-- is implemented in PostgreSQL as
create trigger X before insert on T
for each statement execute procedure F;
create function F ... as $$
begin
   if (C) then ProcCode end if;
end;
$$ language plpgsql;
```

Triggers in PostgreSQL (cont)

Triggers can be activated **BEFORE** or **AFTER** the event.

If activated **AFTER**, the effects of the event are visible:

- **NEW** contains the current value of the altered tuple
- **OLD** contains the previous value of the altered tuple

Sequence of actions during a change:

1. execute any **BEFORE** triggers for this change
2. temporarily make the change and check constraints
3. execute any **AFTER** triggers for this change
4. commit the changes (i.e. make them permanent)

Failure in any of the first three steps rolls back the change.

Triggers in PostgreSQL (cont)

PLpgSQL functions for triggers are defined as

```
-- PostgreSQL 7.3 and later
CREATE OR REPLACE FUNCTION name() RETURNS TRIGGER ...
-- PostgreSQL 7.2
CREATE OR REPLACE FUNCTION name() RETURNS OPAQUE ...
```

There is no restriction on what code can go in the function.

However it must contain one of:

```
RETURN old;     or     RETURN new;
```

depending on which version of the tuple is to be used.

If an exception is raised in the function, no change occurs.

Example PostgreSQL Trigger

Example: ensure that U.S. state names are entered correctly

```
create function checkState() returns trigger as $$
begin
   -- normalise the user-supplied value
   new.state = upper(trim(new.state));
   if (new.state !~ ''^[A-Z][A-Z]$'') then
      raise exception ''State 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;

create trigger checkState before insert or update
on Person for each row execute procedure checkState();
```

Example PostgreSQL Trigger (cont)

Examples of how this trigger would behave:

```
insert into Person
   values('John',...,'Calif.',...);
-- fails with 'Statecode must be two alpha chars'

insert into Person
   values('Jane',...,'NY',...);
-- insert succeeds; Jane lives in New York

update Person
   set town='Sunnyvale',state='CA'
         where name='Dave';
```

```
-- update succeeds; Dave moves to California

update Person
   set state='OZ' where name='Pete';
-- fails with 'Invalid state code OZ'
```

Example PostgreSQL Trigger #2

Implement the Employee update triggers from above in PostgreSQL:

There are three changes that need to be handled:

- case 1: new employee arrives (**`INSERT`**)
- case 2a: employee changes salary (**`UPDATE`**)
- case 2b: employee changes department (**`UPDATE`**)
- case 3: existing employee leaves (**`DELETE`**)

We need a function and trigger for each case.

Example PostgreSQL Trigger #2 (cont)

Case 1: new employee arrives

```
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;
```

Note that the test on **`new.dept`** is not needed; if **`new.dept`** was **`NULL`**, the update would have no effect; having the test does give a marginal performance improvement, by occasionally avoiding an UPDATE.

Example PostgreSQL Trigger #2 (cont)

Case 2: employee changes department/salary

```
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;
```

Example PostgreSQL Trigger #2 (cont)

Case 3: existing employee leaves

```
create function totalSalary3() returns trigger
as $$
begin
```

```
    if (old.dept is not null) then
        update Department
        set    totSal = totSal - old.salary
        where  Department.id = old.dept;
    end if;
    return old;
end;
$$ language plpgsql;
```

Note that the test on **`old.dept`** is not needed; if **`old.dept`** was **`NULL`**, the update would have no effect; having the test does give a marginal performance improvement, by occasionally avoiding an UPDATE.

Example PostgreSQL Trigger #2 (cont)

Finally, we need to define the triggers:

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

create trigger TotalSalary2
```

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

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

Note: all **`after`** triggers because we want to make sure that the changes to the **`Employees`** table are really going to occur.

Trigger Caveat

Mutually recursive triggers can cause infinite loops.

```
create function fixS() returns trigger as $$
    begin update S where a = new.x; return new end;
$$ language plpgsql;

create function fixR() returns trigger as $$
    begin update R where x = new.a; return new end;
$$ language plpgsql;
```

```
create trigger updateR before update on R
for each row execute procedure fixS();

create trigger updateS before update on S
for each row execute procedure fixR();
```

Produced: 13 Sep 2020