PLpgSQL

Procedural Language Extensions for the pgSQL

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
  - massive data, spread over a network
- All are required to assist in application development.

SQL Data Types

- SQL data definition language provides:
  - atomic types: integer, float, character, boolean
  - ability to define tuple types (*create table*)
- SQL also provides mechanisms to define new types:
  - basic types: CREATE DOMAIN
  - tuple types: CREATE TYPE

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 a 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.

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

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

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

What's wrong with SQL? (cont.)

- Solution:

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

What's wrong with SQL? (cont.)

- Some problems:
  - SQL doesn't allow parameterisation (e.g. *AcctNum*)
  - always attempts UPDATE, even when it knows it's invalid
  - always displays balance, even when not changed
- To accurately express the “business logic", we need facilities like conditional execution and parameter passing.

Database programming(cont.)

- 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

    (PL is decoupled from DBMS; most flexible; e.g. Java/JDBC, PHP)
  - embedding SQL into augmented programming languages

    (requires PL pre-processor; typically DBMS-specific; e.g. SQL/C)
  - special-purpose programming languages in the DBMS

    (integrated with DBMS; enables extensibility; e.g. PL/SQL, PLpgSQL)

Database programming(cont.)

- **Recap the example:**
- withdraw *amount* dollars from account *acctNum*
- using a function with parameters *amount* and *acctNum*
- returning two possible text results :
  - 'Insufficient funds' if try to withdraw too much
  - 'New balance *newAmount*' if withdrawal ok
- an obvious side-effect is to change the stored balance
- Requires a combination of
  - SQL code to access the database
  - procedural code to control the process

Database Programming(cont.)

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;
```


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:
  - 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 ( Params )
[ local declarations ]
procedure body ;

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

- Parameters have three modes: IN, OUT, INOUT

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)

SQL Functions

- PostgreSQL Manual: 35.4. Query Language (SQL) Functions
- PostgreSQL allows functions to be defined in SQL

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


SQL Functions(cont.)

- Within the function, arguments are accessed as $1, $2, ...
- Return value: result of the last SQL statement.
- *rettype* can be any PostgreSQL data type (incl tuples,tables).
- Function returning a table: returns setof *TupleType*

SQL Functions(cont.)

- Examples:

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


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
```


SQL Functions(cont.)

- Examples:

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


SQL Functions(cont.)

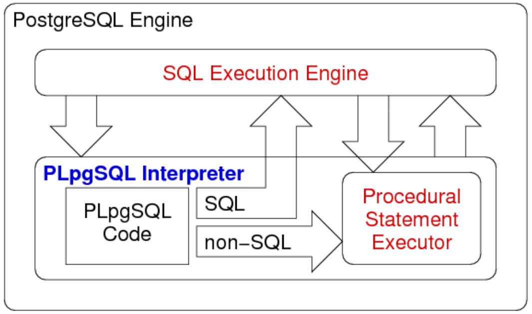
```
-- usage examples
select * from hotelsIn('The Rocks');

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

PLpgSQL

- PostgreSQL Manual: Chapter 40: PLpgSQL
- PLpgSQL = **P**rocedural **L**anguage extensions to **P**ost**greSQL**
- 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(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.

Defining PLpgSQL Functions(cont.)

Recap 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;
```

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

PLpgSQL Function Parameters(cont.)

- Example: old-style function

```
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.

PLpgSQL 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.

PLpgSQL Function Parameters(cont.)

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

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

PLpgSQL Function Parameters(cont.)

- PLpgSQL allows 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
  - void (i.e. no return value)
  - 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)
- 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 PLpgSQL 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 update on R

    for each row execute procedure myCheck ();

Special Data Types

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

  account Accounts % ROWTYPE ;
- Record components referenced via attribute name account.branchName%TYPE

Special 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)
- Example

  quantity INTEGER ;

  start_qty quantity % TYPE ;

  employee Employees % ROWTYPE ;

  name Employees.name % TYPE ;

Control Structures

- Assigment
  - 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
    Satement
END LOOP ;
```

- Example

```
LOOP
    IF count > 0 THEN
            -- some computations
    END IF;
END LOOP;
```


Control Structures(cont.)

- Iteration

```
FOR int_var IN low .. high LOOP
    Satement
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;
```


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" (Exp1, Exp2, ...) as usual
- assign each Expi to corresponding Vari

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

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

- Each Exceptionsi is 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 / y;  ---- y: = # of Tom Jones in Staff Table
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 a variable that can be used to access the result of a particular SQL query
- Cursors move sequentially from row to row (cf., file pointers in C).

Employees

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

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.)

- 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.)

- 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 ;
```


Cursors(cont.)

- 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.

Triggers

- Triggers are
  - procedures stored in the database
  - activated in response to database events (e.g. updates)
- Examples of uses for triggers:
  - maintaining summary data
  - checking schema-level constraints (assertions) on update
  - 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)

Triggers(cont.)

- Consider two triggers and an INSERT statement

```
create trigger X before insert on T Code1;
create trigger Y after insert on T Code2;
insert into T values (a,b,c,...);
```

- Consider two triggers and an UPDATE statement

```
create trigger X before update on T Code1;
create trigger Y after update on T Code2;
update T set b=j,c=k where a=m;
```


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*
[WHEN (*Condition*)]
FOR EACH {ROW|STATEMENT}
EXECUTE PROCEDURE *FunctionName*(*args...*);

Triggers in PostgreSQL(cont.)

- PLpgSQL Functions for Triggers

  CREATE OR REPLACE FUNCTION name () RETURNS TRIGGER ..

- There is no restriction on what code can go in the function.
- However
  - RETURN OLD or RETURN new (depending on which version of the tuple is to be used)
  - Raise an EXCEPTION. In that case, no change occurs

Trigger Example

- Consider a database of people in the USA:

```
create table Person (
        id integer primary key,
        ssn varchar(11) unique,
        ... e.g. family, given, street, town ...
        state char(2), ...
);
create table States (
        id integer primary key,
        code char(2) unique,
        ... e.g. name, area, population, flag ...
);
```

- Constraint: Person.state $\in$ (select code from States), or exists (select id from States where code=Person.state)

Trigger Example(cont.)

- **Example:** ensure that only valid state codes are used:

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

create function checkState() returns trigger as $$
begin
        -- normalise the user-supplied value
        new.state = upper(trim(new.state));
        if (new.state !~ '^[A-Z][A-Z]$') then
                raise exception 'Code must be two alpha chars';
        end if;
        -- implement referential integrity check
        select * from States where code=new.state;
        if (not found) then
                raise exception 'Invalid code %',new.state;
        end if;
        return new;
end;
$$ language plpgsql;
```


Trigger Example(cont.)

- **Example:** department salary totals
- Scenario:

  Employee(id, name, address, dept, salary, ...)

  Department(id, name, manager, totSal, ...)

- An assertion that we wish to maintain:

  Department.totSal =

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

Trigger Example(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 validity after each change.
- With triggers, we have to program each case separately.
- Each program implements updates to *ensure* assertion holds.

Trigger Example(cont.)

- Implement the Employee update triggers from above in PostgreSQL:
- Case 1: new employees arrive

```
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: 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: employees leave

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

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;
```