PLpgSQL

Tuesday, 5 May 2020 6:48 PM

Limitations of Basic SQL
SQL:
data definition language(
atomic types: integer, float, character, boolean
,ability to define tuple types (create table))

constraints, query language ,view.

This is not sufficient to write complete applications.

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

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 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)

PostgreSQL allows functions to be defined in SQL

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

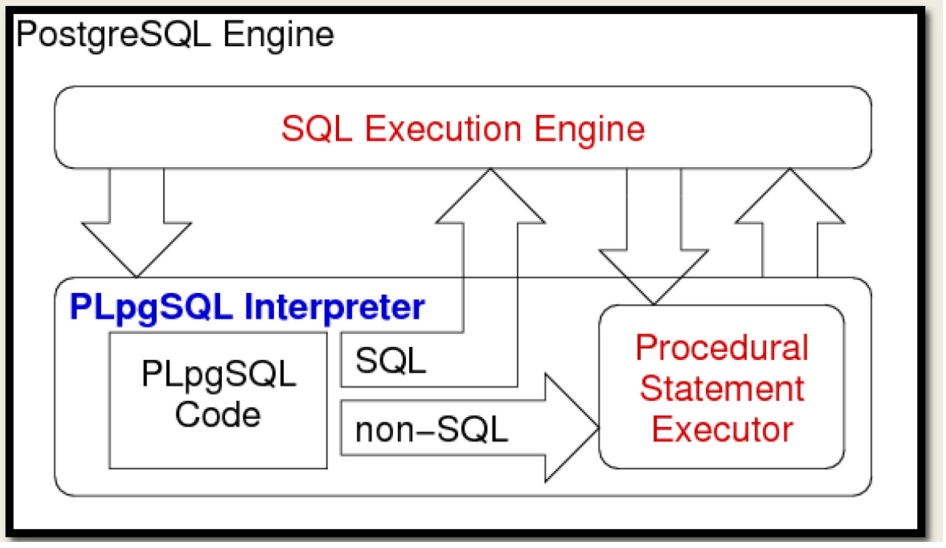
A PostgreSQL-specific language integrating features of:

- 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



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.

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...);
```

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.

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;
```

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;
```

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;
```