

Stored Procedures

- Limitations of Basic SQL
- Extending SQL
- SQL as a Programming Language
- Database Programming
- Stored Procedures
- SQL Functions
- Functions vs Views

❖ 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 provides powerful declarative data extraction mechanisms.

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.

But still do not provide a solution to developing applications.

❖ SQL as a 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
- we need to build a web interface for users to access data

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.

❖ SQL as a Programming Language (cont)

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;
```

❖ SQL as a Programming Language (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, Python)
- 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)

Here we focus on the last: extending DBMS capabilities via programs stored in the DB

❖ 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 are small programs ...

- stored in the database, alongside the stored data
- invoked in SQL queries, or automatically invoked in triggers

SQL/PSM is a standard for stored procedures, developed in 1996. By then, most DBMSs had their own stored procedure languages.

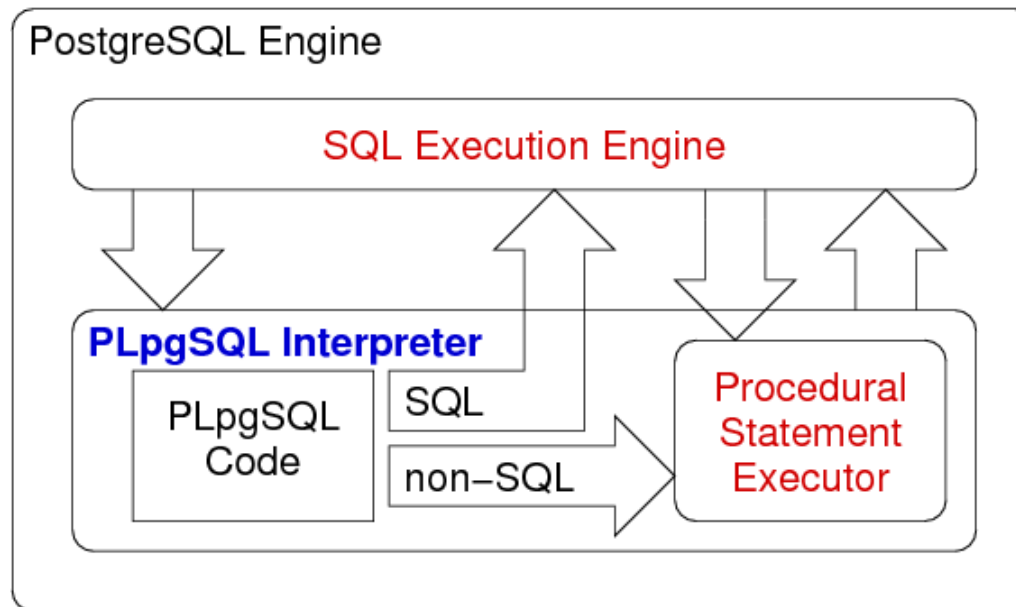
PostgreSQL supports stored procedures in a variety of languages

- PLpgSQL ... PostgreSQL-specific procedural language (cf. Oracle's PL/SQL)
- SQL ... functions that resemble parameterised views
- Python, Perl, Tcl, ... etc.

❖ Stored Procedures (cont)

The PLpgSQL interpreter

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



Embedded in DBMS engine, so efficient to execute with queries

❖ SQL Functions

PostgreSQL allows functions to be defined in SQL

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

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**

Details: PostgreSQL Documentation, Section 37.5

❖ SQL Functions (cont)

Example: info about bars from a given suburb

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

-- usage examples
select * from hotelsIn('The Rocks');
      name      |   addr   | license
-----+-----+-----
Australia Hotel | The Rocks | 123456
Lord Nelson     | The Rocks | 123888

select * from hotelsIn('Randwick');
      name      |   addr   | license
-----+-----+-----
Royal Hotel     | Randwick | 938500
```

❖ SQL Functions (cont)

Example: Name of cheapest beer at each bar

```
create view Cheapest(bar, price) as
select bar, min(price) from Sells group by bar;

select s.*
from   Sells s
where  s.price =
      (select price from Cheapest where bar = s.bar);
```

Could be implemented by defining an SQL function **LowestPriceAt(bar)**

```
create or replace
  function LowestPriceAt(text) returns float
as $$
select min(price) from Sells where bar = $1;
$$ language sql;

select * from Sells where price = LowestPriceAt(bar);
```

❖ Functions vs Views

A parameterless function behaves similar to a view

E.g.

```
create or replace view EmpList
as
select given||' '||family as name,
       street||', '||town as addr
from   Employees;
```

which is used as

```
mydb=# select * from EmpList;
```

❖ Functions vs Views (cont)

Compared to its implementation as a function:

```
create type EmpRecord as (name text, addr text);

create or replace function
    EmpList() returns setof EmpRecord
as $$
select family||' '||given as name,
       street||', '||town as addr
from   Employees
$$ language sql;
```

which is used as

```
mydb=# select * from EmpList();
```


Produced: 5 Oct 2020