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, Python, Java … In particular, using SQL for PLpgSQL function (simply called SQL functions) is also a common usage.

Recall the ValuableEmployees() example from before.

```
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;
```

SQL functions

valuableEmployees() – PLpgSQL version …

```
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;
```

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

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

SQL functions

However, if we want to allow minimum valuable salary to change (i.e., parameterise it), 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 allows SQL statements to have parameters …

SQL functions

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

Differences between SQL and PLpgSQL 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 statement
  (their result is the result of the SQL statement)
- return types can be atomic, tuple, or setof tuples

SQL functions

```
create or replace function
    beerForManf(_manf varchar(30)) returns setof beers
as $$
declare
    e beers%ROWTYPE;
begin
    For e in select * from beers where manf  = _manf
    loop
        return next e;
    end loop;
    return;
end;
$$ language plpgsql;
~
```

Can the above be turned into SQL functions?

Aggregates and User Defined Aggregates

Aggregates reduce a collection of values into a single result.

Often used with GROUP BY to "summarise" each group

Example:

```
R
 a | b | c
---+---+---
 1 | 2 | x
 1 | 3 | y
 2 | 2 | z
 2 | 1 | a
 2 | 3 | b
```

```
select a,sum(b),count(*)
from R group by a
```

```
 a | sum | count
---+-----+-------
 1 |   5 |     2
 2 |   6 |     3
```

Aggregates and User Defined Aggregates

Procedural view of what an aggregate does:

```
AggState = initial state
for each item V {
    # update AggState to include V
    AggState = newState(AggState, V)
}
return final(AggState)
```

All aggregates follow this pattern,

- but differ in initial, final() and newState()

Aggregates and User Defined Aggregates

SQL standard alone does not specify user-defined aggregates.

But PostgreSQL provides a mechanism for creating custom aggregates.

The skeleton of implementing a new aggregate is as explained before … To define a new aggregate, you need to supply PostgreSQL with:

- *BaseType* ... type of input values
- *StateType* ... type of intermediate states
- State Mapping function (how the new state is produced): *sfunc(state,value)* → *newState*
- [optionally] an initial state value (defaults to null)
- [optionally] final function: *ffunc(state)* → *result*

User-defined Aggregates

```
beer=# select * from AGGR;
 first | second
-------+--------
     1 |      2
     3 |      4
     5 |      6
     7 |      8
     9 |     10
(5 rows)
```

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

i.e. $sum2(x,y) = (x_1+y_1) + (x_2+y_2) \ldots (x_n+y_n)$

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

```
AggState = initial state
for each item V {
    # update AggState to include V
    AggState = newState(AggState, V)
}
return final(AggState)
```

```
create function
    AddPair(sum int, _x int, _y int) returns int
as $$
begin return _x+_y+sum; end;    //next state …
$$ language plpgsql;
```

NOTE: AddPair() needs to be defined before aggregate sum2()

User-defined Aggregates

Exercise: Define a concat aggregate that

- takes a column of string values
- returns a comma-separated string of values

For example:

```
select count(*), concat(name) from Employee;
-- returns e.g.
  count |          concat
 -------+-----------------------
      4 | John,Jane,David,Phil
```

Use it to get a list of beers liked by each drinker.

User-defined Aggregates

```
create or replace function
    AddStrName (_t1 text, _t2 text) returns text
as $$ Begin return _t1||','||_t2; end; $$ language plpgsql;

create or replace function
    finalReturnName(_t1 text) returns text
as $$ begin return substr(_t1,2); end;
$$ language plpgsql;

create aggregate concatstr (text) (
    stype      = text,
    initcond   = '',
    sfunc      = AddStrName,
    finalfunc      = finalReturnName
);
```

```
SELECT d.name, concatstr(beer)
FROM drinkers d
    JOIN likes l ON d.name = l.drinker
GROUP BY d.name;
```

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 constraints on table updates
- maintaining summary data (e.g., calculated attributes)
- performing multi-table updates (to maintain constraints)

Triggers

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

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

Triggers can:

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

Triggers

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

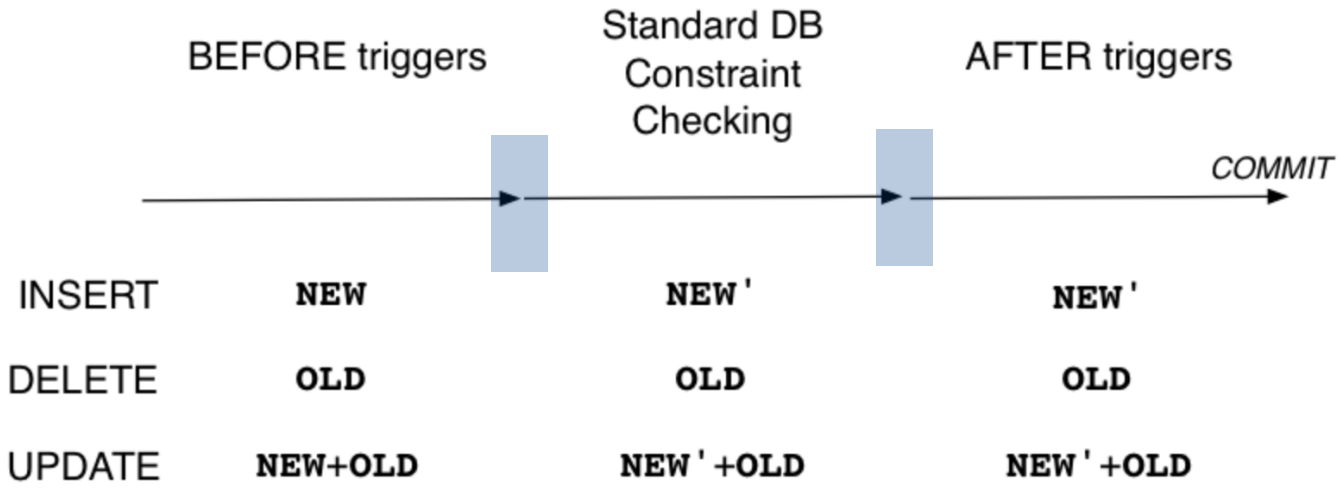
INSERT, DELETE, UPDATE.

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 committed.

Trigger Semantics

Sequence of activities during database update:



Note: BEFORE trigger can modify value of new tuple

Trigger Semantics

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,...);
```

Sequence of events:

1. execute Code1 for trigger X
2. Code1 has access to (a,b,c,...) via NEW
3. Code1 typically checks the values of a,b,c,..
4. Code1 can modify values of a,b,c,.. in NEW
5. DBMS does constraint checking as if NEW is inserted
6. if fails any checking, abort insertion and rollback
7. execute Code2 for trigger Y
8. Code2 has access to final version of tuple via NEW
9. Code2 typically does final checking, or modifies other tables in database to ensure constraints are satisfied

Note: INSERT trigger has no value for OLD

Trigger Semantics

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;
```

Sequence of events:

1. execute Code1 for trigger X
2. Code1 has access to current version of tuple via OLD
3. Code1 has access to updated version of tuple via NEW
4. Code1 typically checks new values of b,c,..
5. Code1 can modify values of b,c,.. in NEW
6. do constraint checking as if NEW has replaced OLD
7. if fails any checking, abort update and rollback
8. execute Code2 for trigger Y
9. Code2 has access to final version of tuple via NEW
10. Code2 typically does final checking, or modifies other tables in database to ensure constraints are satisfied

Note: Update trigger has value for both OLD/NEW

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

Events that might affect the validity of the database

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

A single assertion could check for this after each change. (assertion is not supported in PostgreSQL …)

We can use triggers, but we have to program each case separately.

Each program implements updates to *ensure* the assertion holds. We will basically make sure that each update keeps track of the total salary in the department.

Example Trigger

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

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

Overall syntax:

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

Typical form:

```
-- 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

PostgreSQL trigger syntax does not have conditional activation clause
(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;
```

Can be 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;
```

Example Trigger

ensure that U.S. state names are entered correctly (uses a look up table States)

```
create table states (name varchar(20), code char(2));
insert into states values ('California', 'CA');

create table us_person (name varchar(10), town(10), state char(2));
insert into us_person values ('Dave', 'Sunnyvale', 'CA');

drop function checkState() cascade;

create function checkState() returns trigger as $$
declare
  v char(5);
begin
   -- check the format of new.state
   if (new.state !~ '^[A-Z][A-Z]$') then
      raise exception 'State code must be two alpha chars';
   end if;
   -- implement referential integrity check
   select into v code from States where code=new.state;
   if (not found) then
      raise exception 'Invalid state code %',new.state;
   end if;
   return new;
end;
$$ language plpgsql;

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

Example PostgreSQL Trigger

Examples of how this trigger would behave:

Insert into us_person values ('John', …, 'C'); -- fails, two alpha chars

Insert into us_person values ('Jane', …, 'NY') – format OK, State look up?

Update us_person set town='Sunnyvale', state='CA'
where name = 'Dave' – OK, David is now in CA

Update us_person set state='OZ'
where name = 'Pete' ; -- fail, Invalid state

Example PostgreSQL Trigger #2

Implement the Employee update triggers and totSal example 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.

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

Example PostgreSQL Trigger #2

Case 1: new employee arrives

```
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;
```

```
drop function checkNewEmpSal() cascade;
create function checkNewEmpSal() returns trigger as $$
begin
    update department
    set totalsal = totalsal + new.salary
    where department.id = new.dept;
    return new;
end;
$$ language plpgsql;

create trigger checkNewEmpSal after insert
on employee for each row execute procedure checkNewEmpSal();
```

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;
```

```
drop function checkPayRise() cascade;

create function checkPayRise() returns trigger as $$
declare
  v char(5);
begin
    if old.dept = new.dept and old.salary <> new.salary then
      update department
      set totalsal = totalsal + new.salary - old.salary
      where department.id = new.dept;
    end if;
    return new;
end;
$$ language plpgsql;

create trigger checkPayRise after update
on employee for each row execute procedure checkPayRise();
```

Case 3: employees change departments

BUT WHAT IF WE CAN DO BOTH case 2 and case 3 in one trigger?

```
drop function checkUpdEmpSal() cascade;
create function checkUpdEmpSal() returns trigger as $$
declare
  v char(5);
begin
    if new.dept <> old.dept then -- moving department
      update department
      set totalsal = totalsal + new.salary
      where department.id = new.dept;
      update department
      set totalsal = totalsal - new.salary
      where department.id = old.dept;
    elsif (new.dept = old.dept) and  (new.salary <> old.salary) then
      update department
      set totalsal = totalsal + new.salary - old.salary
      where department.id = new.dept;
    end if;
    return new;
end;
$$ language plpgsql;

create trigger checkNUpdEmpSal after update
on employee for each row execute procedure checkUpdEmpSal();
```

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;
```

```
drop function checkDelEmpSal() cascade;
create function checkDelEmpSal() returns trigger as $$
declare
  v char(5);
begin
      update department
      set totalsal = totalsal - old.salary
      where department.id = old.dept;
      return old;
end;
$$ language plpgsql;

create trigger checkNDelEmpSal after delete
on employee for each row execute procedure checkDelEmpSal();
```

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();
```