>>

Triggers

- Triggers
- Trigger Semantics
- Triggers in PostgreSQL
- Trigger Example #1
- Trigger Example #2

^ >>

❖ 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)

Some typical variations within 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: **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

<< ^ >>

❖ Trigger Semantics

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

If activated **`BEFORE`**, can affect the change that occurs:

- **`NEW`** contains "proposed" value of changed tuple
- modifying **`NEW`** causes a different value to be placed in DB

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

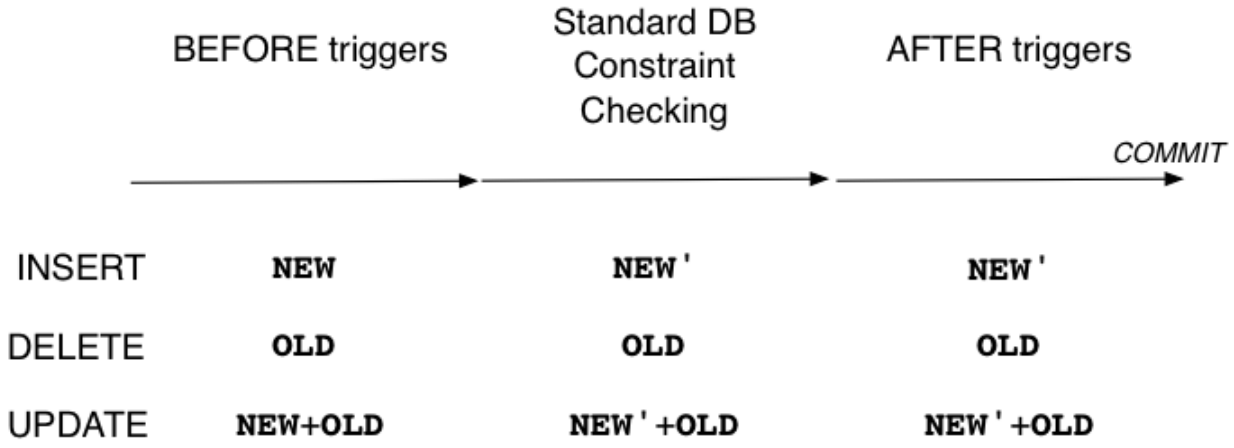
- **`NEW`** contains the current value of the changed tuple
- **`OLD`** contains the previous value of the changed tuple
- constraint-checking has been done for **`NEW`**

Note: **`OLD`** does not exist for insertion; **`NEW`** does not exist for deletion.

<< ^ >>

❖ Trigger Semantics (cont)

Sequence of activities during database update:



Reminder: **BEFORE** trigger can modify value of new tuple

❖ Trigger Semantics (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,...);
```

Sequence of events:

- execute **`Code1`** for trigger **`X`**
- code has access to **`(a,b,c,...)`** via **`NEW`**
- code typically checks the values of **`a,b,c,..`**
- code can modify values of **`a,b,c,..`** in **`NEW`**
- DBMS does constraint checking as if **`NEW`** is inserted
- if fails any checking, abort insertion and rollback
- execute **`Code2`** for trigger **`Y`**
- code has access to final version of tuple via **`NEW`**

- code typically does final checking, or modifies other tables in database to ensure assertions are satisfied

Reminder: there is no **`OLD`** tuple for an **`INSERT`** trigger.

❖ Trigger Semantics (cont)

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:

- execute **Code1** for trigger **X**
- code has access to current version of tuple via **OLD**
- code has access to updated version of tuple via **NEW**
- code typically checks new values of **b,c,..**
- code can modify values of **a,b,c,..** in **NEW**
- do constraint checking as if **NEW** has replaced **OLD**
- if fails any checking, abort update and rollback
- execute **Code2** for trigger **Y**
- code has access to final version of tuple via **NEW**

- code typically does final checking, or modifies other tables in database to ensure constraints are satisfied

Reminder: both **`OLD`** and **`NEW`** exist in UPDATE triggers.

❖ Trigger Semantics (cont)

Consider two triggers and an DELETE statement

```
create trigger X before delete on T Code1;
create trigger Y after delete on T Code2;
delete from T where a=m;
```

Sequence of events:

- execute **`Code1`** for trigger **`X`**
- code has access to **`(a,b,c,...)`** via **`OLD`**
- code typically checks the values of **`a,b,c,..`**
- DBMS does constraint checking as if **`OLD`** is removed
- if fails any checking, abort deletion (restore **`OLD`**)
- execute **`Code2`** for trigger **`Y`**
- code has access to about-to-be-deleted tuple via **`OLD`**
- code typically does final checking, or modifies other tables in database to ensure constraints are satisfied

Reminder: tuple **`NEW`** does not exist in DELETE triggers.

<< ^ >>

❖ 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)

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

However a **BEFORE** function must contain one of:

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

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

It can also return **NULL**; no further checks are done and the operation fails.

If **BEFORE** trigger returns **OLD**, no change occurs.

If exception is raised in trigger function, no change occurs.

❖ Trigger Example #1

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` ∈ `(select code from States)`**, or
`exists (select id from States where code=Person.state)`

<< ^ >>

❖ Trigger Example #1 (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 #1 (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'
```

❖ Trigger Example #2

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 * from Department d
       where  d.totSal <> (select sum(e.salary)
                           from Employee e
                           where e.dept = d.id)
    )
)
```

<< ^ >>

❖ Trigger Example #2 (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.

<< ^ >>

❖ Trigger Example #2 (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 #2 (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 #2 (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;
```

Produced: 26 Oct 2021