Introduction to Triggers using SQL

Kristian Torp

Department of Computer Science
Aalborg University
www.cs.aau.dk/~torp
torp@cs.aau.dk

November 24, 2011

Center for Data-intensive Systems

daisy.aau.dk

Outline

Learning Outcomes

Goals

- Understand basic trigger mechanism
- Understand pros and cons of triggers
- See procedural code in a DBMS

Why?

- Widely used in database applications
- Widely supported in most DBMSs

Note

- Concepts presented are general
- Code presented is PostgreSQL specific
  - One is Oracle specific because not supported on PostgreSQL

Two Minute Discussion

Usage of Triggers

- Have you use triggers before?
- Have you used triggers for deriving temporal information?
- Does your organization have any policies wrt. triggers?

Outline

Case Study: Handling Football Players

Note

- Keep track of the players in all football clubs
- Keep a history of where each player has been playing

Case Study: Handling Football Players

Note

- Keep track of the players in all football clubs
- Keep a history of where each player has been playing

At Time 10

```
-- AaB buys the player Wurtz
insert into player values 'Wurtz', 'AaB');
```

Player Table

Name	Club
Wurtz	AaB

Log Table

Case Study: Handling Football Players

Note

- Keep track of the players in all football clubs
- Keep a history of where each player has been playing

At Time 10

```
-- AaB buys the player Wurtz
insert into player values 'Wurtz', 'AaB');
```

Player Table

Name	Club
Wurtz	AaB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	null

Case Study: Handling Football Players

Note

- Keep track of the players in all football clubs
- Keep a history of where each player has been playing

At Time 10

```
-- AaB buys the player Wurtz
insert into player values 'Wurtz', 'AaB');
```

Player Table

Name	Club
Wurtz	AaB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	null

Note

- All values are copied from the table player to the table log
- A value is automatically provided for the column start_date
- A value cannot be provided for the column stop_date

Case Study: Handling Football Players, cont.

At Time 12

```
-- AaB buys the player Caca
insert into player values 'Caca', 'AaB');
```

Case Study: Handling Football Players, cont.

At Time 12

```
-- AaB buys the player Caca
insert into player values 'Caca', 'AaB');
```

Player Table

Name	Club
Wurtz	AaB
Caca	AaB

Log Table

Case Study: Handling Football Players, cont.

At Time 12

```
-- AaB buys the player Caca
insert into player values 'Caca', 'AaB');
```

Player Table

Name	Club
Wurtz	AaB
Caca	AaB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	null
Caca	AaB	12	null

Case Study: Handling Football Players, cont.

At Time 12

```
-- AaB buys the player Caca
insert into player values 'Caca', 'AaB');
```

Player Table

Name	Club
Wurtz	AaB
Caca	AaB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	null
Caca	AaB	12	null

Note

- Similar to the first insert
- Start date is the current-time, stop date is unknown/not specified/TBD

Case Study: Handling Football Players, cont.

At Time 15

```
-- AaB sells Wurtz to FCK
update player set club = 'FCK' where name = 'Wurtz';
```

Player Table

Name	Club
Wurtz	FCK
Caca	AaB

Log Table

Case Study: Handling Football Players, cont.

At Time 15

```
-- AaB sells Wurtz to FCK
update player set club = 'FCK' where name = 'Wurtz';
```

Player Table

Name	Club
Wurtz	FCK
Caca	AaB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	15
Caca	AaB	12	null
Wurtz	FCK	15	null

Case Study: Handling Football Players, cont.

At Time 15

```
-- AaB sells Wurtz to FCK
update player set club = 'FCK' where name = 'Wurtz';
```

Player Table

Name	Club
Wurtz	FCK
Caca	AaB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	15
Caca	AaB	12	null
Wurtz	FCK	15	null

Note

- The table player is naturally updated
- The column stop_date for the first Wurtz row is updated in the log table
- A new row is entered into the log table

Case Study: Handling Football Players, cont.

At Time 25

```
-- AaB sells Caca to OB
update player set club = 'OB' where name = 'Caca';
```

Player Table

Name	Club
Wurtz	FCK
Caca	OB

Log Table

Case Study: Handling Football Players, cont.

At Time 25

```
-- AaB sells Caca to OB
update player set club = 'OB' where name = 'Caca';
```

Player Table

Name	Club
Wurtz	FCK
Caca	OB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	15
Caca	AaB	12	25
Wurtz	FCK	15	null
Caca	OB	25	null

Case Study: Handling Football Players, cont.

At Time 25

```
-- AaB sells Caca to OB
update player set club = 'OB' where name = 'Caca';
```

Player Table

Name	Club
Wurtz	FCK
Caca	OB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	15
Caca	AaB	12	25
Wurtz	FCK	15	null
Caca	OB	25	null

Note

- Similar to the first update

Case Study: Handling Football Players, cont.

At Time 27

```
-- AaB buys Wurtz back from FCK
update player set club = 'AaB' where name = 'Wurtz';
```

Case Study: Handling Football Players, cont.

At Time 27

```
-- AaB buys Wurtz back from FCK
update player set club = 'AaB' where name = 'Wurtz';
```

Player Table **Log Table**

Name	Club
Wurtz	AaB
Caca	OB

Case Study: Handling Football Players, cont.

At Time 27

```
-- AaB buys Wurtz back from FCK
update player set club = 'AaB' where name = 'Wurtz';
```

Player Table

Name	Club
Wurtz	AaB
Caca	OB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	15
Caca	AaB	12	25
Wurtz	FCK	15	27
Caca	OB	25	null
Wurtz	AaB	27	null

Case Study: Handling Football Players, cont.

At Time 27

```
-- AaB buys Wurtz back from FCK
update player set club = 'AaB' where name = 'Wurtz';
```

Player Table

Name	Club
Wurtz	AaB
Caca	OB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	15
Caca	AaB	12	25
Wurtz	FCK	15	27
Caca	OB	25	null
Wurtz	AaB	27	null

Note

- A new Wurtz row is started in the log table
- Still, similar to the previous updates
  - Basically a delete followed by an insert

Case Study: Handling Football Players, cont.

At Time 62

```
-- Wurtz retires as an active football player
delete from player name = 'Wurtz';
```

Case Study: Handling Football Players, cont.

At Time 62

```
-- Wurtz retires as an active football player
delete from player name = 'Wurtz';
```

Player Table **Log Table**

Name	Club
Caca	OB

Case Study: Handling Football Players, cont.

At Time 62

```
-- Wurtz retires as an active football player
delete from player name = 'Wurtz';
```

Player Table

Name	Club
Caca	OB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	15
Caca	AaB	12	25
Wurtz	FCK	15	27
Caca	OB	25	null
Wurtz	AaB	27	62

Case Study: Handling Football Players, cont.

At Time 62

```
-- Wurtz retires as an active football player
delete from player name = 'Wurtz';
```

Player Table

Name	Club
Caca	OB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	15
Caca	AaB	12	25
Wurtz	FCK	15	27
Caca	OB	25	null
Wurtz	AaB	27	62

Note

- The Wurtz row is deleted from the player table
- The value column stop_date is update for the last Wurtz row

Quiz: Handling Football Players

Final Tables

Player Table

Name	Club
Caca	OB

Log Table

Name	Club	start_date	stop_date
Wurtz	AaB	10	15
Caca	AaB	12	25
Wurtz	FCK	15	27
Caca	OB	25	null
Wurtz	AaB	27	62

Questions

- How to you identify the active players in the log table?
- What is the interpretation of the null values in the log table?
- What is the primary key of the log table?
- In how many clubs can a single player be active at once?
- When are rows deleted from the log table?

Outline

Audit Trail Example

```
create table player(
    player_id      int            primary key,
    player_name    varchar(50)    not null,
    date_of_birth  date,
    club           varchar(50)    not null);
```

Audit Trail Example

```
create table player(
    player_id      int           primary key,
    player_name    varchar(50)   not null,
    date_of_birth  date,
    club           varchar(50)   not null);

create table player_log(
    player_id      int           not null,
    player_name    varchar(50)   not null,
    date_of_birth  date,
    club           varchar(50)   not null,
    start_date     date          not null,
    stop_date      date);
```

Note

- The primary keys are not the same in the two tables
- The stop_date column is nullable

Insert Trigger

```
create or replace function f_player_ins() returns trigger as $$
begin
    insert into player_log values(
          new.player_id,      new.player_name,
          new.date_of_birth, new.club,
          current_date,       null);
     return new;
end;
$$ language 'plpgsql';
```

Insert Trigger

```
create or replace function f_player_ins() returns trigger as $$
begin
    insert into player_log values(
        new.player_id,      new.player_name,
        new.date_of_birth, new.club,
        current_date,       null);
    return new;
end;
$$ language 'plpgsql';

create trigger player_ins
    after insert on player
    for each row execute procedure f_player_ins();
```

Insert Trigger

```
create or replace function f_player_ins() returns trigger as $$
begin
    insert into player_log values(
        new.player_id,      new.player_name,
        new.date_of_birth, new.club,
        current_date,       null);
    return new;
end;
$$ language 'plpgsql';

create trigger player_ins
    after insert on player
    for each row execute procedure f_player_ins();
```

Note

- The new syntax
- The null is inserted into the stop_date column
- The current_date is ”give me the current date”

Let Dreams Come Trough

```
insert into player values
    (101, 'Wurtz', '1987-03-03', 'AaB');
insert into player values
    (102, 'Messi', '1988-02-13', 'AaB');
update player
    set club = 'Barcelona'
    where player_name = 'Messi';
delete from player
    where player_id = 101;
```

Note

- The “clients” cannot see the derived information

Delete Trigger

```
create or replace function f_player_del() returns trigger as $$
begin
    update player_log
    set    stop_date = current_date
    where  player_id = old.player_id
    and    stop_date is null;
    return old;
end;
$$ language 'plpgsql';
```

Delete Trigger

```
create or replace function f_player_del() returns trigger as $$
begin
    update player_log
    set    stop_date = current_date
    where  player_id = old.player_id
    and    stop_date is null;
    return old;
end;
$$ language 'plpgsql';

create trigger player_del
    after delete on player
    for each row execute procedure f_player_del();
```

Delete Trigger

```
create or replace function f_player_del() returns trigger as $$
begin
    update player_log
    set    stop_date = current_date
    where  player_id = old.player_id
    and    stop_date is null;
    return old;
end;
$$ language 'plpgsql';

create trigger player_del
    after delete on player
    for each row execute procedure f_player_del();
```

Note

- This trigger is only fired for delete statements

Update Trigger

```
create or replace function f_player_upd() returns trigger as $$
begin
    -- stop old
    update player_log
    set    stop_date = current_date
    where  player_id = old.player_id
    and    stop_date is null;
    -- start new
    insert into player_log values(
        new.player_id,      new.player_name,
        new.date_of_birth,  new.club,
        current_date,       null);
    return new;
end;
$$ language 'plpgsql';
```

Update Trigger

```
create or replace function f_player_upd() returns trigger as $$
begin
    -- stop old
    update player_log
    set    stop_date = current_date
    where  player_id = old.player_id
    and    stop_date is null;
    -- start new
    insert into player_log values(
        new.player_id,      new.player_name,
        new.date_of_birth,  new.club,
        current_date,       null);
    return new;
end;
$$ language 'plpgsql';

create trigger player_upd
    after update on player
    for each row execute procedure f_player_upd();
```

Outline

Overview

- Triggers are executed implicitly
  - When insert, update, or delete statements are executed
- Similar to a stored procedure, i.e., code
  - Can make call-outs to procedural code
- Connected to a table
  - In some DBMSs also on a view
- Triggers are side effects
  - Normally considered very bad in software engineering
- Triggers not part of SQL-92 first introduced in SQL-1999.
  - Many DBMS have supported triggers for much longer therefore limited standard compliance

Transition Tables

- A set of “internal” tables that the DBMS uses to keep track of modifications made by a transaction

Four transition tables for each “real” table R during the execution of a transaction T_i

$R_{inserted}$	Contains the rows inserted into R during T_i
$R_{deleted}$	Contains the rows deleted from R during T_i
$R_{updatedold}$	Contains the values of updated rows before T_i
$R_{updatednew}$	Contains the values of updated rows after T_i

$$R_{new} = R_{old} \setminus R_{deleted} \setminus R_{updatedold} \cup R_{inserted} \cup R_{updatednew}$$

Outline

Usage

- To generate derived values
  - Sum of amount of order lines
- To create an audit trail
  - Think STASI
- To enforce complex business rules
  - When a customer buys goods for more than 300$ then give a 10% discount
- To generate statistics
  - How often is a table modified
- To provide event logging
  - Each time a new house is inserted in the for_sale table notify potential customers

Another Example: Compute Derived Values I

User Requirements

- Wants to store order and order lines
- The total amount of an order is derived from order lines
- The total amount is often queried

Solution

- Store the derived information total amount with the order
  - Makes it fast to look-up orders based on total amount
- Use triggers to automatically derived the total amount
  - Convenient for the customers

Another Example: Compute Derived Values II

```
create table oorder(
    oorder_id      int             primary key,
    customer_name  varchar2(50)    not null,
    amount         number(10,2)    default 0 not null,
        constraint amount_gt_zero check(amount > 0)
            deferrable initially immediate
);

create table oorder_line(
    oorder_id  int           not null,
    line_no    int           not null check (line_no > 0),
    dsc        varchar2(50)  not null,
    quantity   int           not null check (quantity > 0),
    price_each number(6,2)   not null check (price_each > 0.0),
    constraint ol_pk primary key (oorder_id, line_no),
    constraint ol_o_fk foreign key (oorder_id)
        references oorder(oorder_id)
);
```

Another Example: Compute Derived Values III

```
create or replace trigger set_amount
    after insert or update or delete on oorder_line
for each row
declare
  val number(10,2) := 0;
  oid int;
begin
  if inserting then
    val := :new.quantity* :new.price_each;
    oid := :new.oorder_id;
  elsif updating then
    val := :new.quantity * :new.price_each -
           :old.quantity * :old.price_each;
    oid := :new.oorder_id;
  elsif deleting then
    val := 0 - :old.quantity * :old.price_each;
    oid := :old.oorder_id;
  end if;
```

Another Example: Compute Derived Values IV

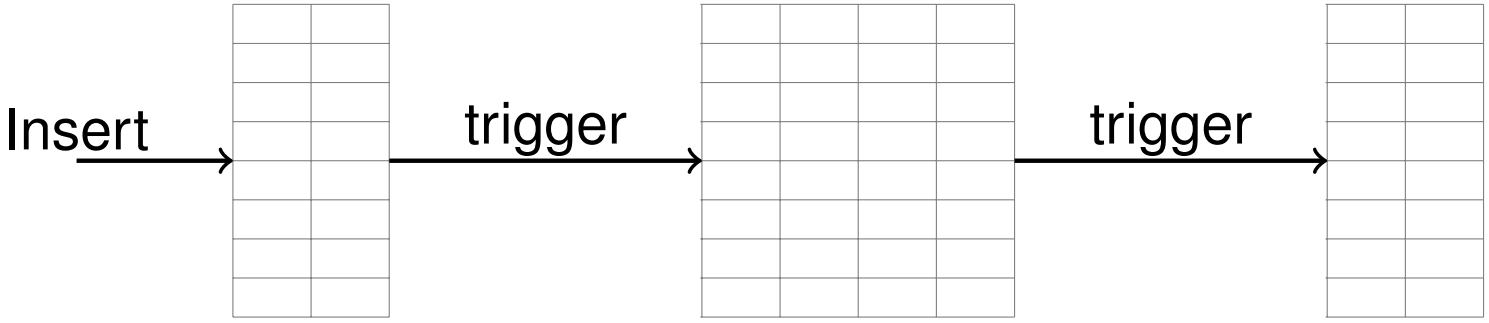
```
    update oorder
        set amount = amount + val
        where oorder_id = oid;
end;
```

A Bad Example

```
create or replace trigger set_amount2
    after insert on oorder
for each row
declare
    tmp_amount number(10,2);
begin
   select sum(quantity * price_each)
   into   tmp_amount
   from   oorder_line
   where  oorder_id = :new.oorder_id;

   update oorder
     set amount = tmp_amount
     where oorder_id = :new.oorder_id;
end;
```

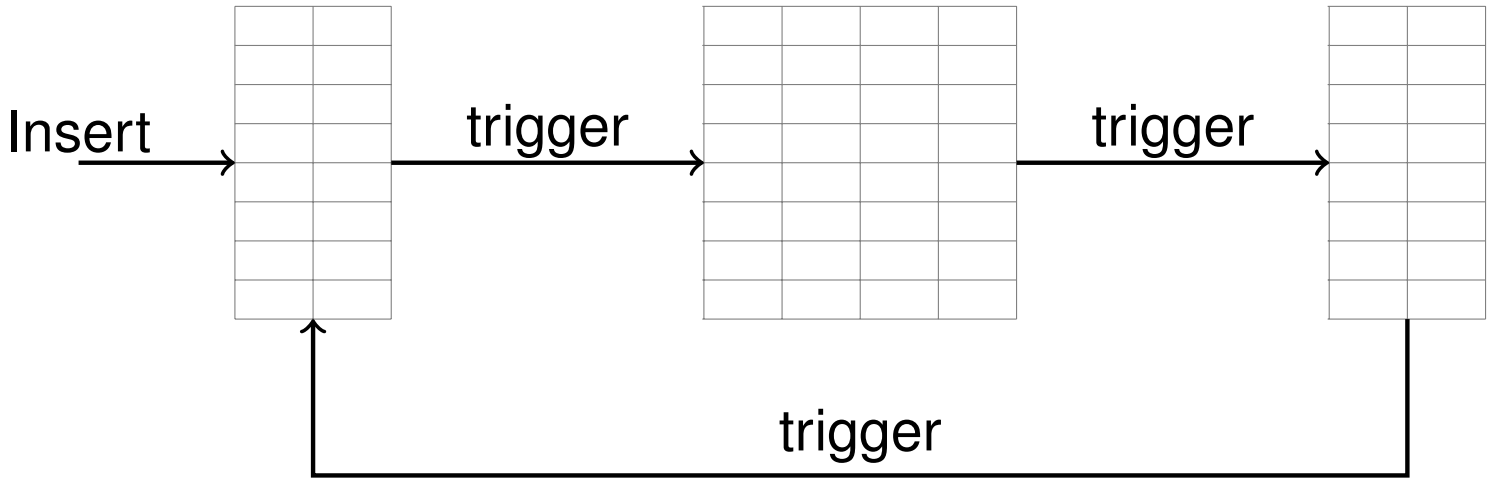
Mutating Triggers



Note

- A trigger can make an insert, this new insert may fire a trigger, etc
- Cannot have cycles will cause a mutating trigger
- Causes a runtime error

Mutating Triggers



Note

- A trigger can make an insert, this new insert may fire a trigger, etc
- Cannot have cycles will cause a mutating trigger
- Causes a runtime error

Correct Usage

- Can have multiple triggers on a table
  - Order of execution cannot be specified
- Can be hard to understand a database schema with many triggers
- A trigger cannot modify the table that it is associated with
  - May cause an infinite loop of trigger executions

Correct Usage

- Can have multiple triggers on a table
  - Order of execution cannot be specified
- Can be hard to understand a database schema with many triggers
- A trigger cannot modify the table that it is associated with
  - May cause an infinite loop of trigger executions

Not for Integrity Constraints

Do not use triggers for enforcing referential integrity!

Trigger Parts

- Event
  - on update of <table name> ...
- Condition
  - Must evaluate to true
- Action
  - Code that is executed

Two Minute Discussion

Triggers vs. Application Code

- Why not move trigger logic to applications?
  - Discuss the pros and cons of this
- Would you like to have a trigger on select statements
  - When and where could it be useful?

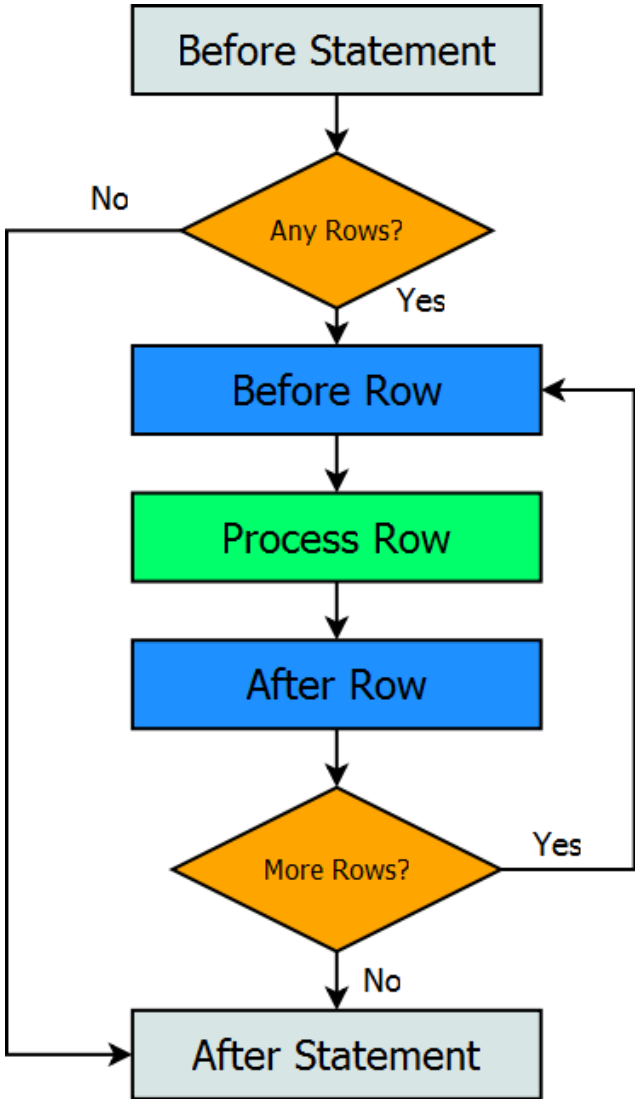
Outline

Trigger Types I

$$\left\{ \begin{array}{c} \text{row} \\ \text{statement} \end{array} \right\} \times \left\{ \begin{array}{c} \text{before} \\ \text{after} \end{array} \right\}$$

- A total of four trigger types
- Some DBMSs support INSTEAD OF triggers
  - Very powerful, e.g., to make view updateable or hide legacy code
- Before triggers can be used for preconditions
- After triggers can be used for postconditions

Trigger Types II



- Statement level triggers always fires
- Row trigger fires only if rows are modified

Row Triggers: Setup Table

Row Triggers

- Executed once for each row modified by triggering statement
- If no rows modified ⇒ trigger is not executed

Example (Create Table)

```
create table x (i int primary key,
                j int not null);
```

Example (Populate Table)

```
insert into x values (1,1);
insert into x values (2,2);
insert into x values (3,3);
commit;
```

Before Row Trigger

Example (Stored Procedure)

```
-- update row before trigger
create or replace function f_x_upd_before_row()
returns trigger as $$
begin
    raise notice 'before row';
    return new;
end;
$$ language 'plpgsql';
```

Example (Before Row Trigger)

```
create trigger x_upd_before_row
    before update on x
for each row execute procedure f_x_upd_before_row();
```

After Row Trigger

Example (Stored Procedure)

```
-- update row after trigger
create or replace function f_x_upd_after_row()
returns trigger as $$
begin
    raise notice 'after row';
    return new;
end;
$$ language 'plpgsql';
```

Example (After Row Trigger)

```
create trigger x_upd_after_row
    after update on x
for each row execute procedure f_x_upd_after_row();
```

Statement Triggers

Statement-Level Triggers

- Executed once for each triggering statement
- Executed even if no rows are modified

Statement Triggers

Statement-Level Triggers

- Executed once for each triggering statement
- Executed even if no rows are modified

Example (Stored Procedure)

```
create or replace function f_x_upd_before_stmt()
returns trigger as $$
begin
    raise notice 'before stmt';
    return new;
end;
$$ language 'plpgsql';
```

Statement Triggers

Statement-Level Triggers

- Executed once for each triggering statement
- Executed even if no rows are modified

Example (Stored Procedure)

```
create or replace function f_x_upd_before_stmt()
returns trigger as $$
begin
    raise notice 'before stmt';
    return new;
end;
$$ language 'plpgsql';
```

Example (Before Statement Trigger)

```
create trigger x_upd_before_stmt
    before update on x
for each statement execute procedure f_x_upd_before_stmt();
```

After Statement Trigger

Example (Stored Procedure)

```
create or replace function f_x_upd_after_stmt()
returns trigger as $$
begin
    raise notice 'after stmt';
    return new;
end;
$$ language 'plpgsql';
```

After Statement Trigger

Example (Stored Procedure)

```
create or replace function f_x_upd_after_stmt()
returns trigger as $$
begin
    raise notice 'after stmt';
    return new;
end;
$$ language 'plpgsql';
```

Example (After Statement Trigger)

```
create trigger x_upd_after_stmt
    after update on x
for each statement execute procedure f_x_upd_after_stmt();
```

Other Types of Triggers

System Event Triggers

- At database start up or shut down
- When a DBMS fails

User-Event Ttriggers

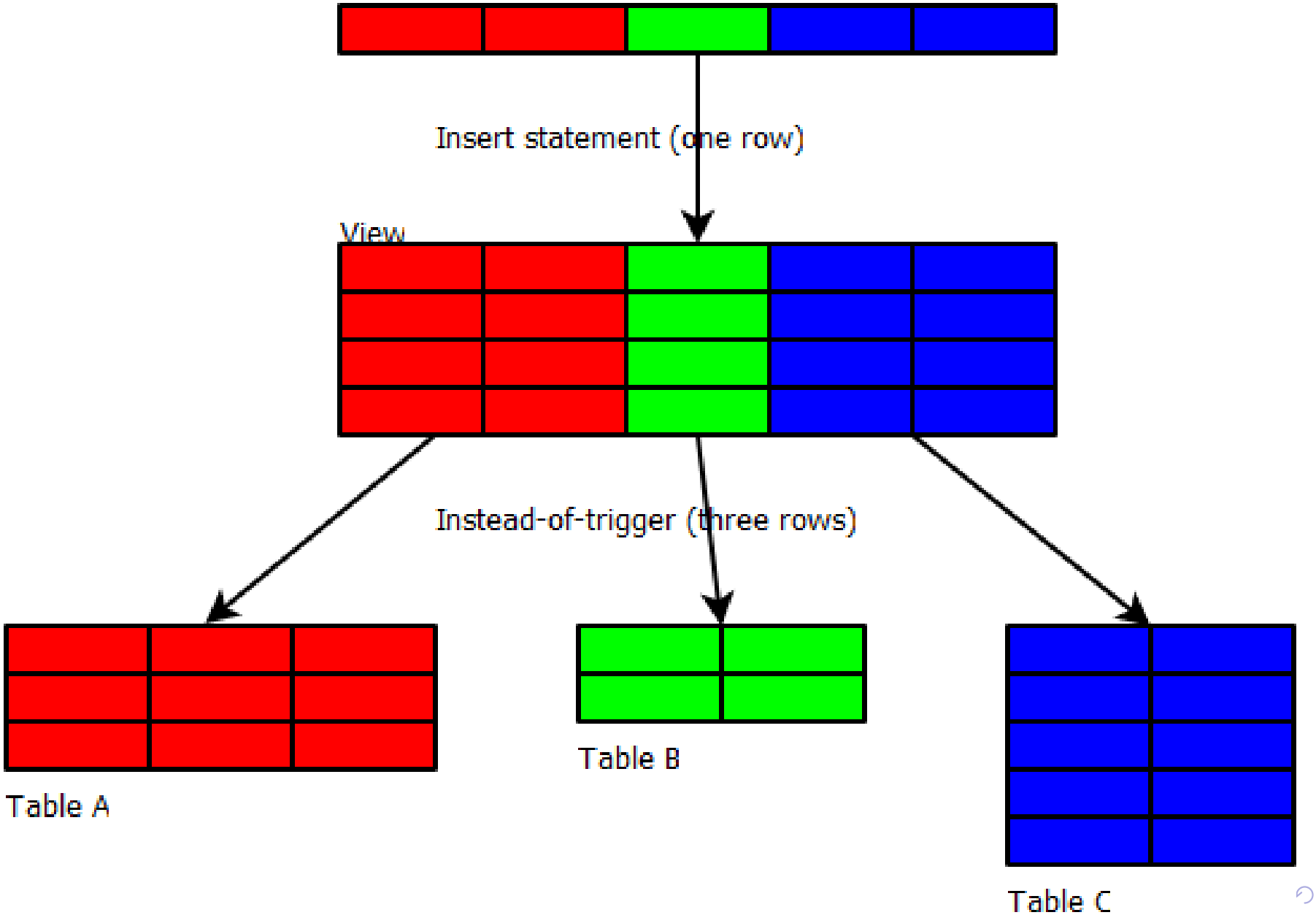
- On logon and logoff
- When DML statement executed
- When DDL statement executed

Note

- These types of triggers are not part of the SQL standard!

Outline

Introduction



Very Strong Concept

- Can make all views updateable
- Can make modifications easier to understand
- Can wrap ugly database design, by nice design
- Can make the changes work for multiple programming languages

Mini University Schema

Customer Requirements

- Wants to have a readable access to students status without joins
  - A student name must have the column name name
  - A student status must the column name status
- Wants to be able to modify a students status using strings

Challenges

- name is a reserved word
- status is also a name of an exiting table
- Currently needs to join to get readable status of students
- Currently needs to memorize the int values used for status

Solutions

- Create a view
- Use instead-of-triggers on view

Create the View

Example (The View)

```
create or replace view student_status as
    select stu.s_name as "name", sta.dsc as "status"
    from   student stu, status sta
    where  stu.stat_id = sta.stat_id
```

Note

- The name and status columns are in quotes
- A simple view statement

Insert Function I

```
create or replace function f_ins_student_status()
    returns trigger as $$
declare
    v_stat_id status.stat_id%type;
    v_sid     student.sid%type;
    v_no      numeric;
begin
    -- check that the status column exists in the status table
    begin
        select sta.stat_id
        into   v_stat_id
        from   status sta
        where  sta.dsc = lower(new."status");
    exception
        when no_data_found then
            null; -- Need to raise an error!!!!
    end;
```

Insert Function II

```
    -- find if there are any students with the same name
    select count(stu.sid)
    into   v_no
    from   student stu
    where  stu.s_name = new."name";
    if v_no > 0 then
        null; -- raise an error
    end if;
    -- find a new student id
    select coalesce(max(stu.sid),0)
    into   v_sid
    from   student stu;
    v_sid := v_sid + 1;
    insert into student
        values (v_sid, new."name", v_stat_id);
    return new;
end;
$$ language plpgsql;
```

Insert Instead-Of-Trigger

Example (Create Trigger)

```
create trigger ins_student_status
    instead of insert on student_status
    for each row execute procedure f_ins_student_status();
```

Note

- All inserts on student_status will call stored procedure
- View is now updateable
  - Only insert supported
- Very few limit on what can be done in stored procedure

Delete Function

```
create or replace function f_del_student_status()
    returns trigger as $$
declare
    v_stat_id status.stat_id%type;
begin
    begin
        select sta.stat_id
        into   v_stat_id
        from   status sta
        where  sta.dsc = lower(old."status");
    exception
        when no_data_found then
            null; -- Need to raise an error!!!!
    end;

    delete from student
        where s_name  = old."name"
        and   stat_id = v_stat_id;
    return new;
end;
$$ language 'plpgsql';
```

Delete Instead-Of-Trigger

Example (Create Trigger)

```
create trigger del_student_status
    instead of delete on student_status
    for each row execute procedure f_del_student_status();
```

Note

- All delete on student_status will call stored procedure
- Similar idea as for the insert instead-of trigger
- Trigger for update can be defined in a similar way

Use the Updateable View

Example (Update Through View)

```
-- insert through the view
insert into student_status
values ('Curt', 'active');

-- delete through the view
delete from student_status
    where name = 'Ann';
```

Note

- View now behave (almost) like a table

Summary: Instead-Of Triggers

Summary

- A very powerful concept
- Generally well-supported most DBMSs
- Old or ugly design can removed and old application logic retained

Note

- Insert, update, and delete can be combined in a single stored procedure
  - Should make sense!
- Too many triggers makes the database very hard to maintain!

Outline

Enabling and Disabling Triggers

Example (Disable Triggers)

```
-- disable single trigger
alter table x disable trigger x_upd_after_row;
-- disable all triggers on a single table
alter table x disable trigger all;
```

Enabling and Disabling Triggers

Example (Disable Triggers)

```
-- disable single trigger
alter table x disable trigger x_upd_after_row;
-- disable all triggers on a single table
alter table x disable trigger all;
```

Example (Enable Triggers)

```
-- enable single trigger
alter table x enable trigger x_upd_after_row;
-- enable all triggers on a single table
alter table x enable trigger all;
```

Enabling and Disabling Triggers

Example (Disable Triggers)

```
-- disable single trigger
alter table x disable trigger x_upd_after_row;
-- disable all triggers on a single table
alter table x disable trigger all;
```

Example (Enable Triggers)

```
-- enable single trigger
alter table x enable trigger x_upd_after_row;
-- enable all triggers on a single table
alter table x enable trigger all;
```

Note

- Be careful, know why a trigger is disabled!

Trigger Metadata

Example (Find Information on All Triggers)

```
select *
from   information_schema.triggers;
```

Note

- Cannot see if trigger is enabled/disabled

Trigger Metadata

Example (Find Information on All Triggers)

```
select *
from   information_schema.triggers;
```

Note

- Cannot see if trigger is enabled/disabled

Example (Find All Disabled Triggers)

```
select p.*
from   pg_trigger as p
where  p.tgenabled = 'D'
```

Note

- This is PostgreSQL specific

Dropping Triggers

Example (Find Information on All Triggers)

```
-- Drop row trigger on table
drop trigger x_upd_after_row on x;

-- Drop instead-of trigger on view
drop trigger ins_student_status on student_status;
```

Note

- The triggers are now removed from the database schema!
- There is no syntax for dropping all triggers on a table!

Summary: Trigger Management

Summary

- Triggers can be enabled/disabled
- When table/view dropped all triggers automatically dropped
- A view cannot always altered therefore triggers on views cannot be altered

Note

- Disabling triggers can lead to hard to find bugs
- Many DBMSs have vendor specific extension for triggers

Outline

Use or Avoid Triggers?

Use Triggers

- For any type of logging
  - Also called auditing
- For deriving information
- For advanced checking that cannot be implemented using integrity constraints
- For automating tasks associated with modification statements
- For simple data replication
- To wrap bad database design
  - That must be retained of backwards compatibility reasons

Avoid Triggers

- When a declarative statement can be used
  - Triggers are procedural
- When foreign key or check statement can be used

Summary

- E-C-A rules (event-condition-action)
- Triggers are called implicitly
  - Side-effects
- {before, after} $\times$ {row, statement}
- Instead-of triggers interesting for hiding legacy design Very powerful concept
- Avoid transaction handling in triggers (commit or rollback)
- Do not overuse the number of triggers
  - Can make it very hard to find out what a system does
- Many vendor specific extensions for trigger functionality
  - Because very late before added to the standard
  - Core idea large overlap between DBMSs

Additional Information

- PL/pgSQL `www.postgres.cz/index.php/PL/pgSQL_(en)`
  - How to write the trigger stored procedures
- Triggers in MySQL `http://forge.mysql.com/wiki/Triggers`
  - Good overview, well explained, linear readable document
- Exploring SQL Server Triggers `msdn.microsoft.com/en-us/magazine/cc164047.aspx`
  - Good introduction, quite readable
- Triggers in Oracle `www.java2s.com/Tutorial/Oracle/0560_Trigger/Catalog0560__Trigger.htm`
  - Only code, many details, Oracle specific

Note

- Cannot find a good general introduction to triggers
  - Send me an email if you are aware of such an introduction

Two-Minute Digesting

Questions

- Why can you not specify the order in which triggers execute?
- Why can triggers not be stand-alone database objects?
- Why are triggers part of the transaction?
- You have the same logic that must be executed by a trigger and sometimes explicitly what do you do?