CS213

Principles of Database Systems(H)

Chapter 9

Procedure and Triger

Shiqi YU 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

9.1 Procedure

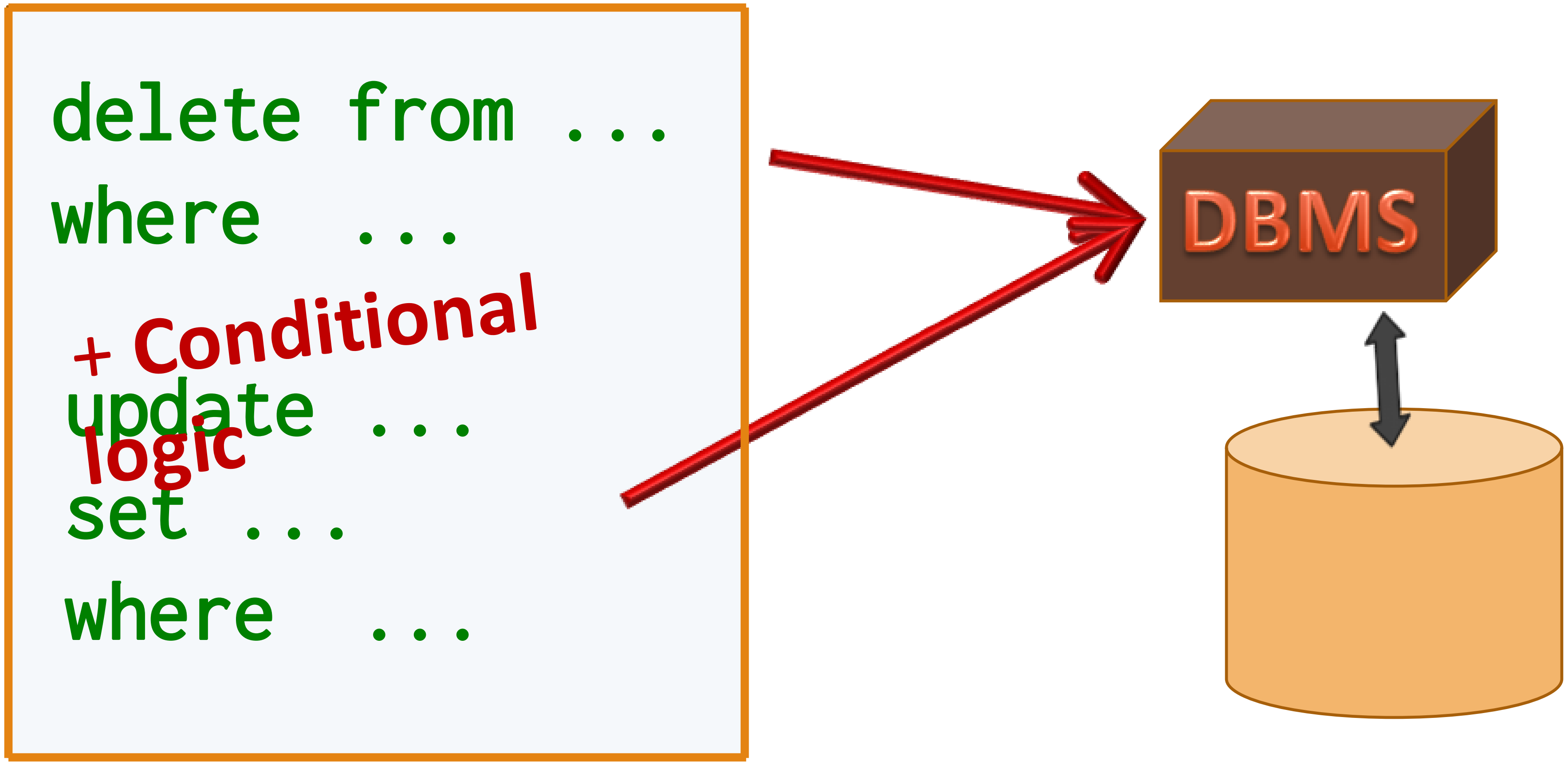
Shiqi Yu 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

Procedures

I have talked about functions, which return a value, let's talk about procedures, which don't (PostgreSQL only knows about functions, but it has void functions)



```
delete from ...
where   ...

update ...
set ...
where   ...
```

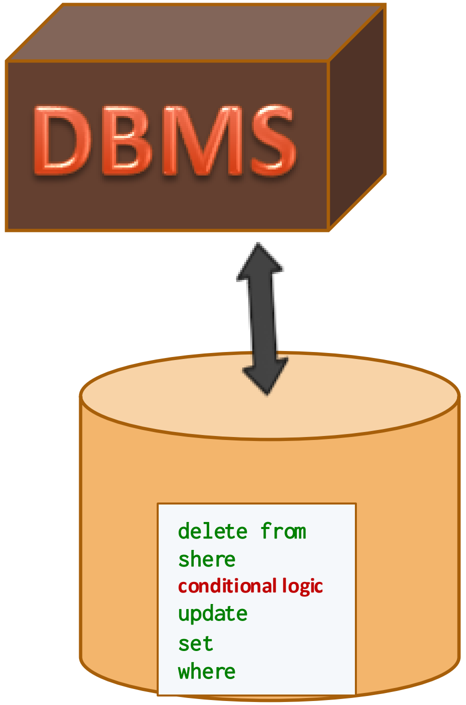
Transactions (i.e. an "everything succeeds or fails" business operation) demand several steps, and may require some conditional logic.

create procedure myproc

```
delete from ...
where   ...

update ...
set ...
where   ...
```

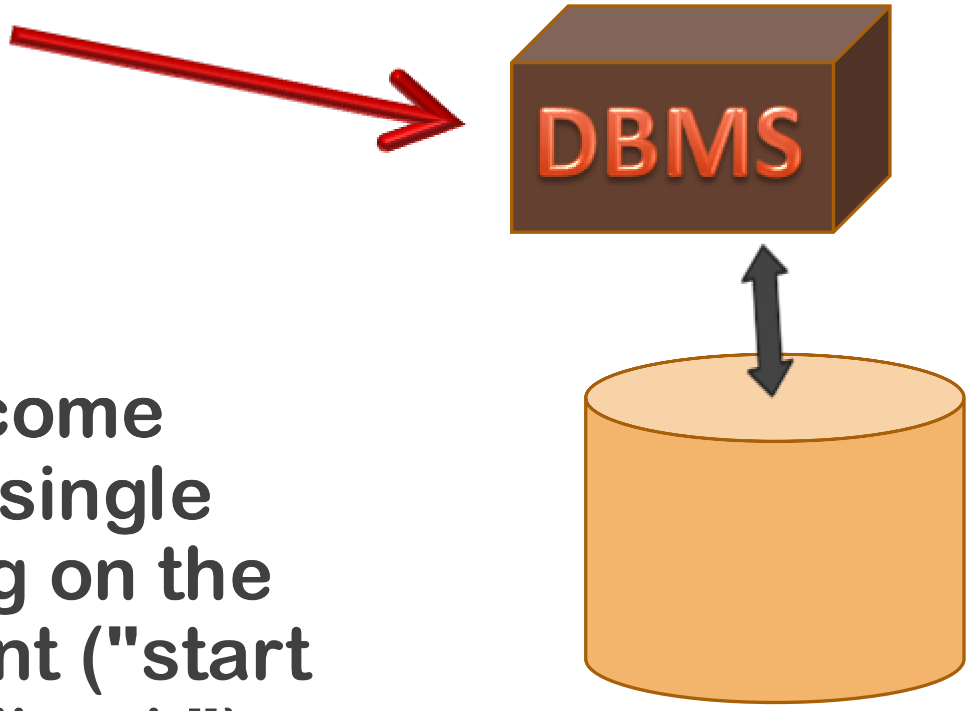
+ Conditional logic



Stored procedure

It makes sense to turn them into a single unit, a procedure that will be stored in the database.

```
execute myproc
```



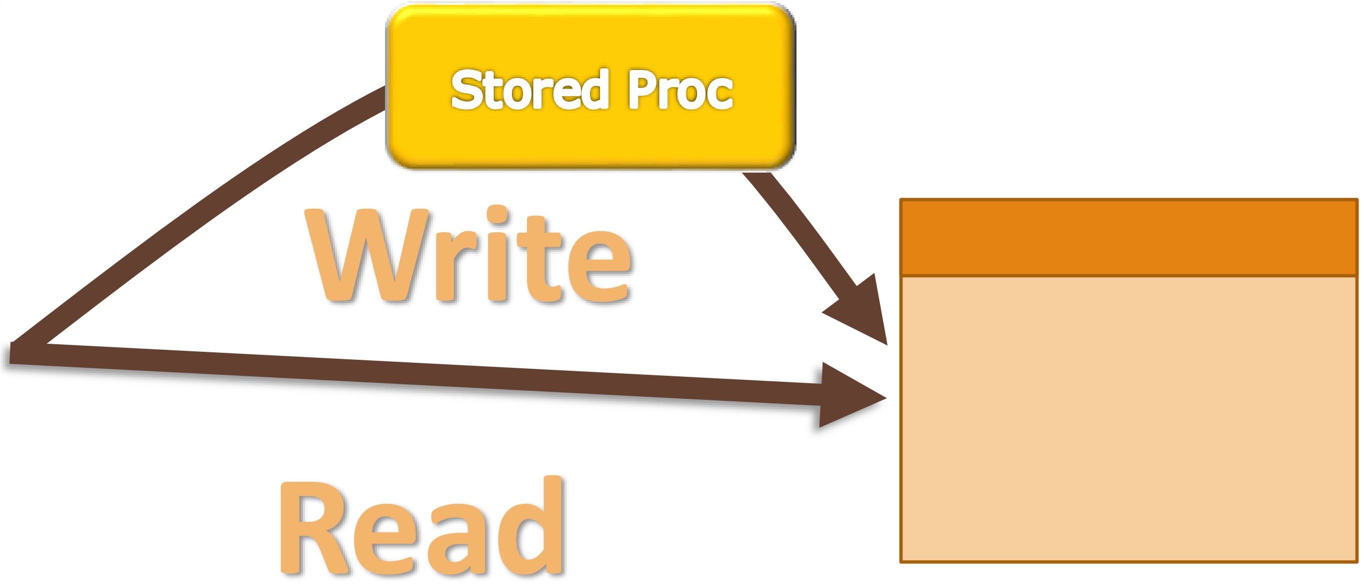
Instead of issuing several SQL statements, checking their outcome and so far, we can then issue a single command to execute everything on the server. Transaction management ("start transaction" and "commit"/"rollback") can be performed inside the procedure or outside it.

ONE
call

There are many benefits to the approach. First of all calling the procedure is a single interaction with the database. When the database is located on a remote server (think "the cloud") you aren't going to waste time chatting over the network with the remote server.

Another significant benefit is security. We haven't talked about it yet, but you can prevent users from modifying data otherwise than by calling carefully written and well tested procedures.

Security



Adding a film to the database is a rather painful exercise. Let's do it with a procedure. The choice of parameters isn't very good but simpler.

Movie registration

Title

Year

Country Name

Director

Actor1

Actor2

Not too good

first name

surname

```
select country_code from countries
...
insert into movies
...
select peopleid from people
...
insert into credits
...
select peopleid from people
...
insert into credits
...
select peopleid from people
...
insert into credits
...
```

get value of movieid

director

actor1

actor2

Lots of things to do

MINIMIZE the number of STATEMENTS

First of all, if we want to be relatively efficient we should try to minimize our interactions with the database. Running a stored procedure on the database is of course much better than issuing statements from afar, but context switches are always costly.

```
select country_code from countries
...
insert into movies
...
select peopleid from people
...
insert into credits
...
select peopleid from people
...
insert into credits
...
select peopleid from people
...
insert into credits
...
```

```
insert into movies ...
select country_code, ...
from countries
...

insert into credits ...
select peopleid, 'D', ...
from people
...

insert into credits ...
select peopleid, 'A', ...
from people
...

insert into credits ...
select peopleid, 'A', ...
from people
...
```

INSERT ... SELECT ... is far better than SELECT followed by an INSERT.

But the three last statements are basically the same one.

```
insert into movies ...
select country_code, ...
from countries
...

insert into credits ...
select peopleid, 'D', ...
from people
...

insert into credits ...
select peopleid, 'A', ...
from people
:::

insert into credits ...
select peopleid, 'A', ...
from people
...
```

```
from (select director, 'D', ...
        union all
        select actor1, 'A', ...
        union all
        select actor2, 'A', ...) a
      inner join people
```

Check one row inserted
if not, generate error

Check rows inserted

Here is what we'll do.

```
create function movie_registration
           (p_title        varchar,
            p_country_name varchar,
            p_year         int,
            p_director_fn  varchar,
            p_director_sn  varchar,
            p_actor1_fn    varchar,
            p_actor1_sn    varchar,
            p_actor2_fn    varchar,
            p_actor2_sn    varchar)
returns void
as $$
declare
  n_rowcount int;
  n_movieid int;
  n_people int;
begin
  insert into movies(title, country, year_released)
    select p_title, country_code, p_year
      from countries
      where country_name = p_country_name;
```

And here is how we can write it with PostgreSQL.

```
insert into movies(title, country, year_released)
  select p_title, country_code, p_year
   from countries
   where country_name = p_country_name;
get diagnostics n_rowcount = row_count;
if n_rowcount = 0
then
   raise exception 'country not found in table COUNTRIES';
end if;
n_movieid := lastval();
select count(surname)
into n_people
from (select p_director_sn as surname
      union all
      select p_actor1_sn as surname
      union all
      select p_actor2_sn as surname) specified_people
where surname is not null;
```

```
insert into credits(movieid, peopleid, credited_as)
select n_movieid, people.peopleid, provided.credited_as
    from (select coalesce(p_director_fn, '*') as first_name,
                p_director_sn as surname,
                'D' as credited_as
          union all
          select coalesce(p_actor1_fn, '*') as first_name,
                 p_actor1_sn as surname,
                 'A' as credited_as
          union all
          select coalesce(p_actor2_fn, '*') as first_name,
                 p_actor2_sn as surname,
                 'A' as credited_as) provided
      inner join people
        on people.surname = provided.surname
          and coalesce(people.first_name, '*') = provided.first_name
where provided.surname is not null;
get diagnostics n_rowcount = row_count;
if n_rowcount != n_people
  then
     raise exception 'Some people couldn''t be found';
end if;
end;
$$ language plpgsql;
```

Check whether we found everybody

In PostgreSQL you can call the procedure interactively by calling it from a SELECT statement (that will return nothing). Other products use "call", "execute", and so on. You can also call a procedure from within another procedure by using "perform".

```
select movie_registration('The Adventures of Robin Hood',
                           'United States', 1938,
                           'Michael', 'Curtiz',
                           'Errol', 'Flynn',
                           null, null);
```

When call from another procedure

```
perform movie_registration('The Adventures of Robin Hood',
                           'United States', 1938,
                           'Michael', 'Curtiz',
                           'Errol', 'Flynn',
                           null, null);
```

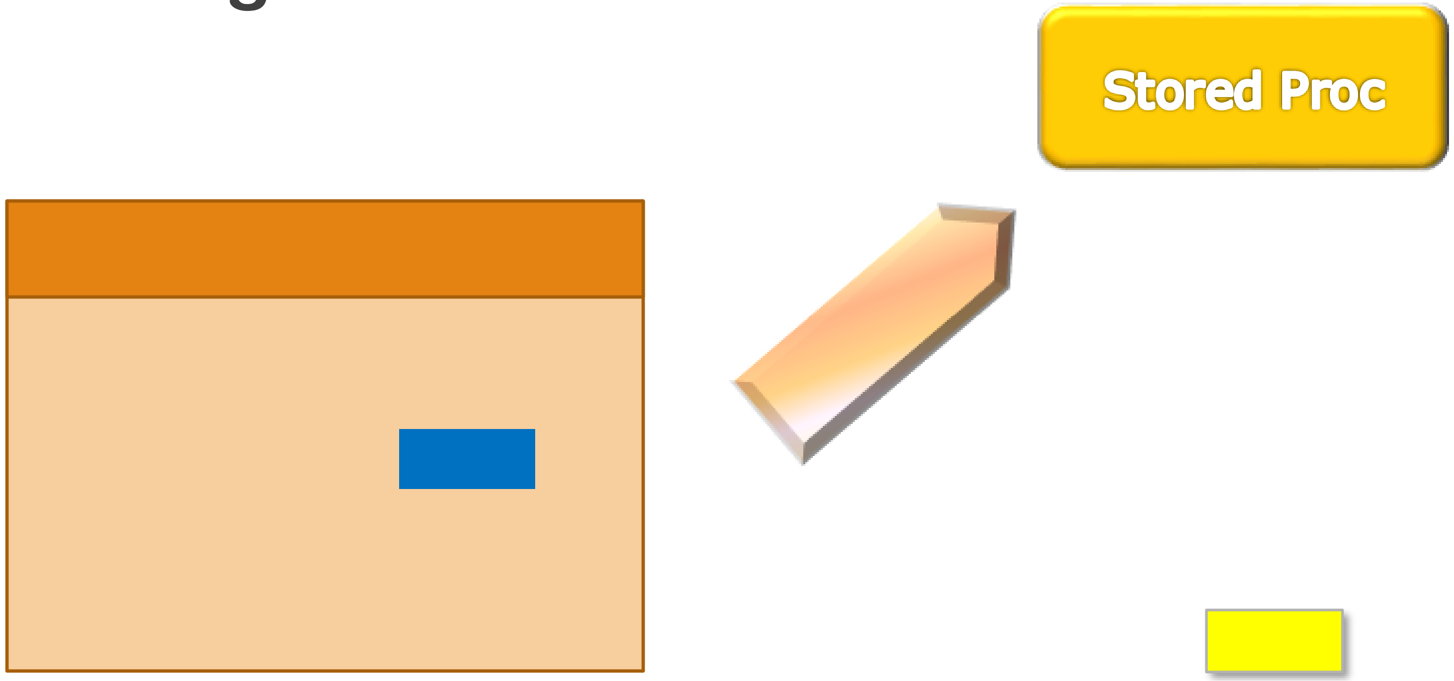
9.2 Trigger

Shiqi Yu 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

You can attach to a table actions that will be executed automatically whenever the data in the table changes.

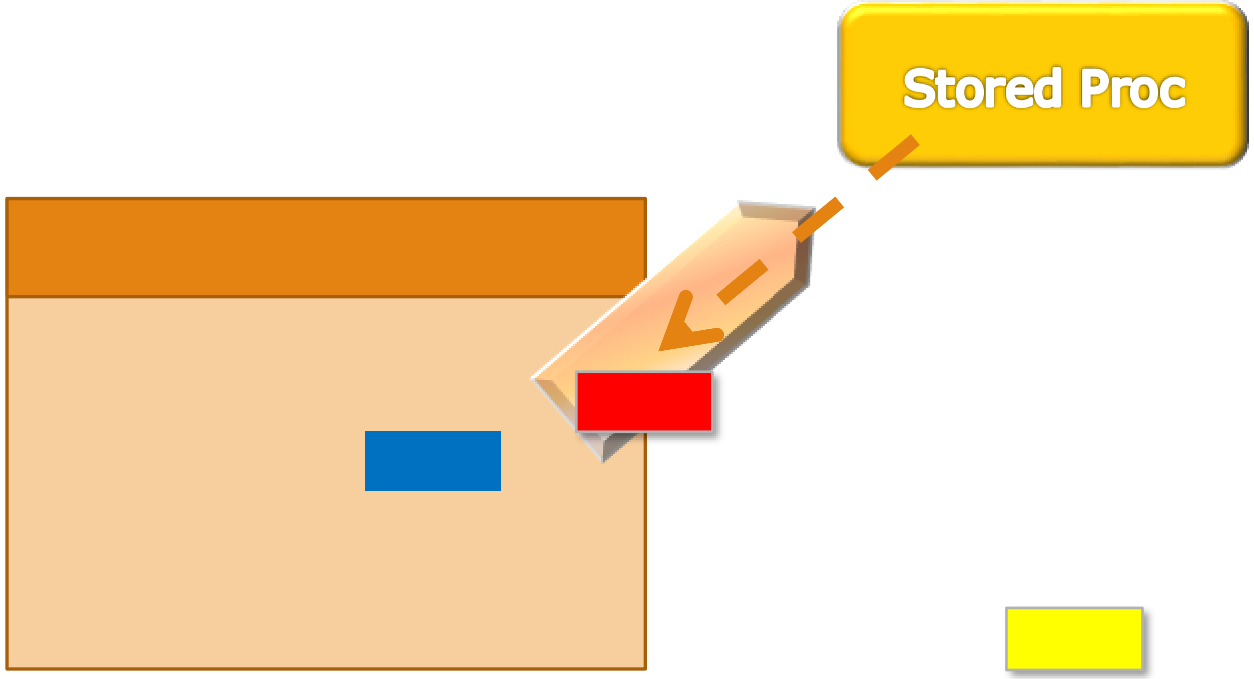


Note: a SELECT will never trigger anything. Only write operations do.

PURPOSE

There are several purposes for triggers, some of which are more commendable than others. That said, we aren't living in an ideal world and there are cases when they can be useful for fixing things which are badly done in a program for which you haven't the source code.

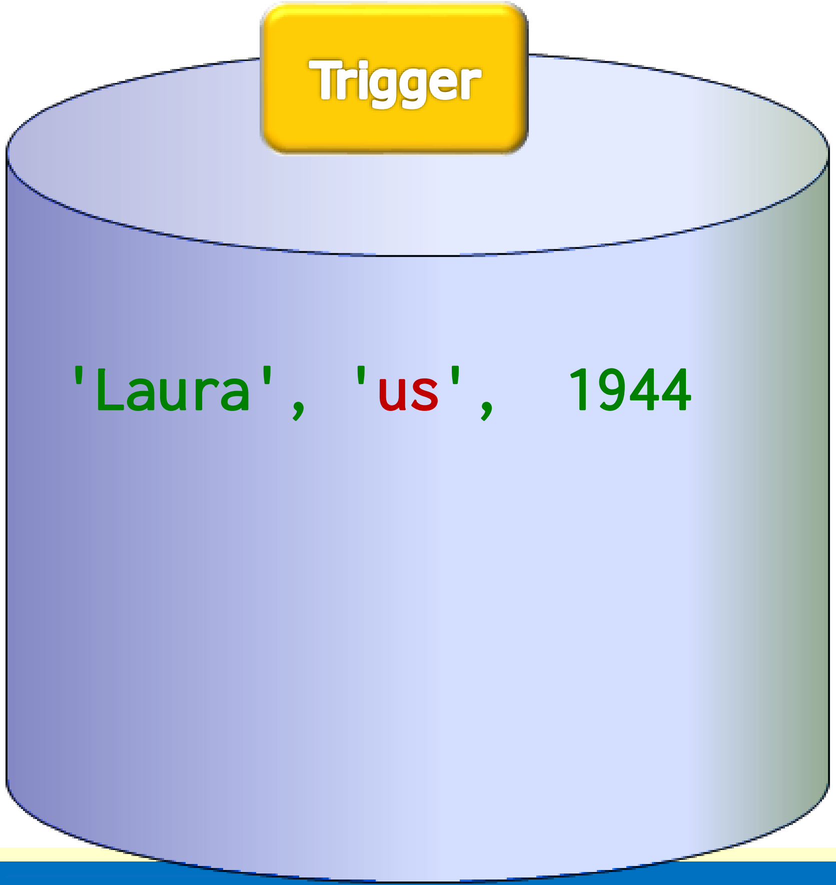
❶ Modify input on the fly



One thing that triggers may be used for is changing input on the fly. For instance, you want to make sure that data is always in lowercase but the data entry program doesn't enforce it, and you have no access to its source code. A trigger can force the case.

Note!

```
insert into movies(title, country, year_released)
values ( 'Laura', 'US',  1944                     )
```

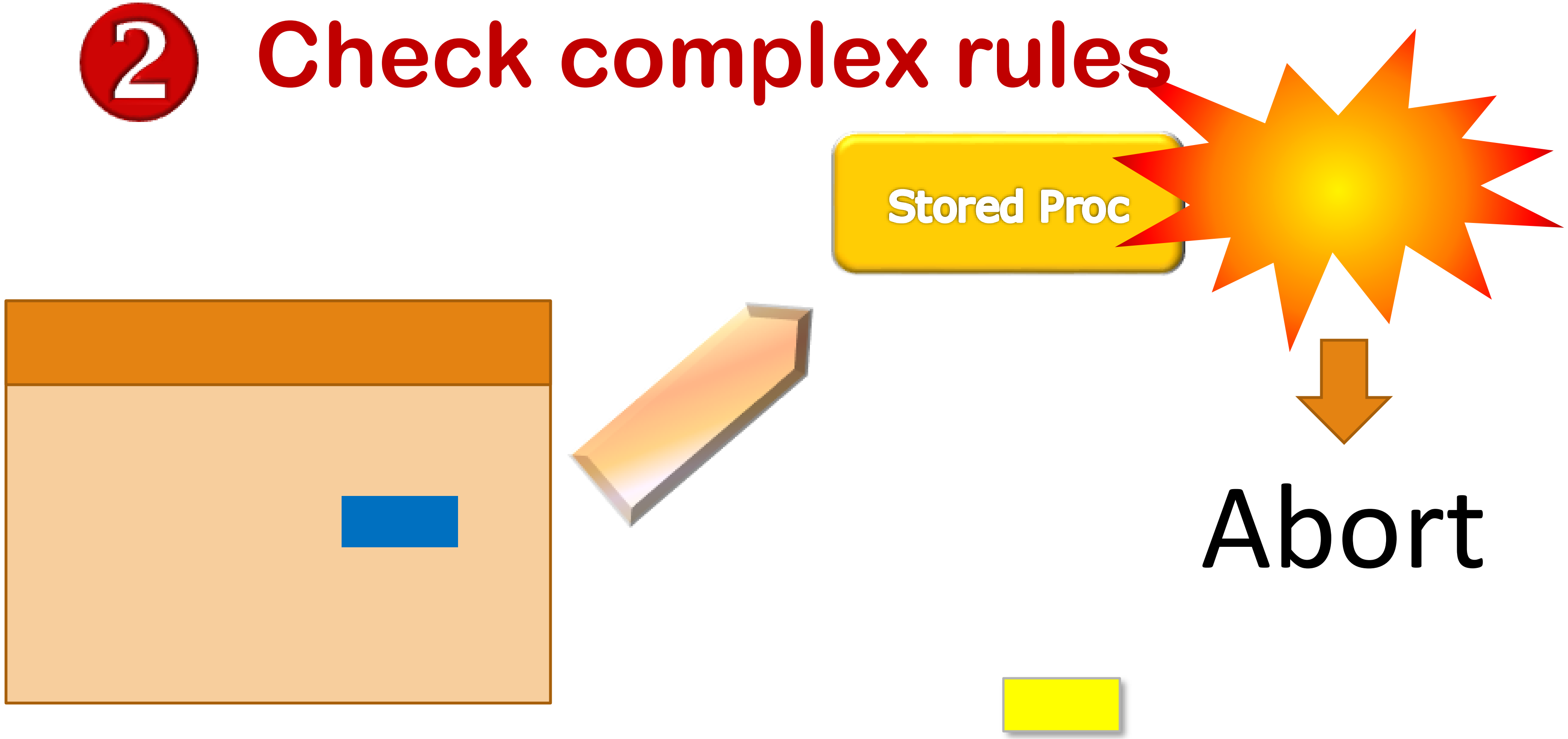


Trigger

```
'Laura', 'us',  1944
```

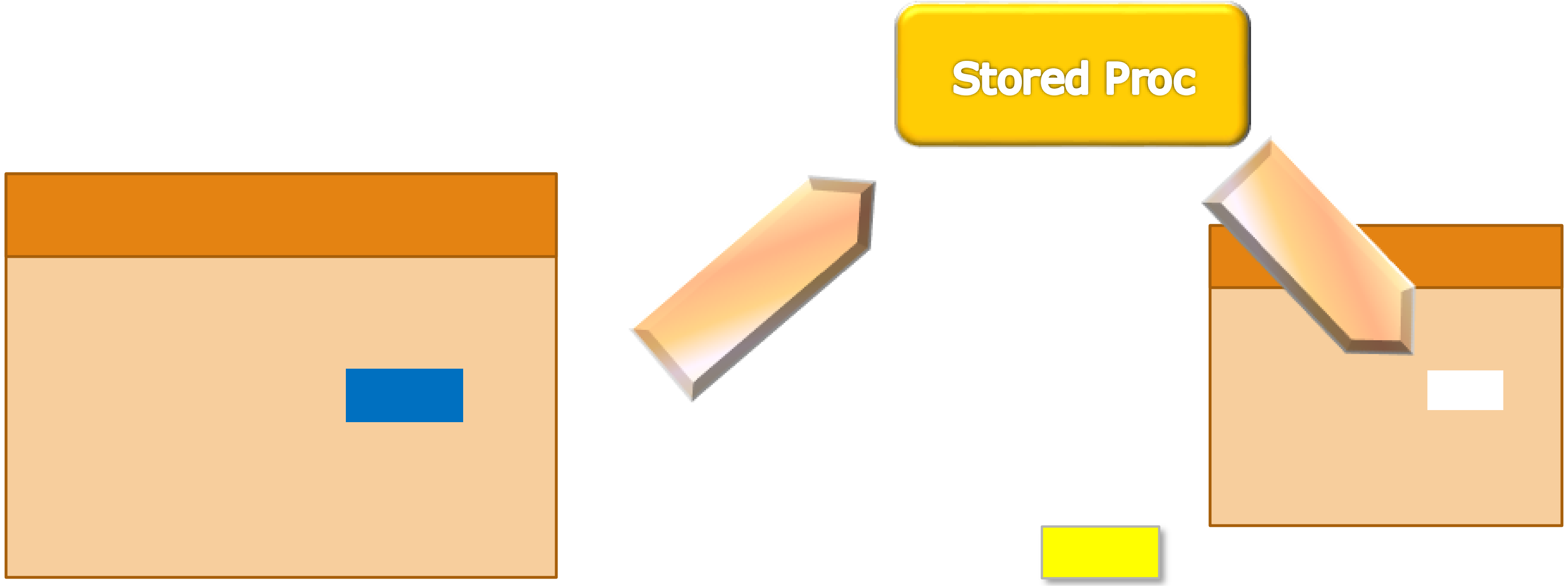
Much better to have the transformation in input programs ....

❷ Check complex rules



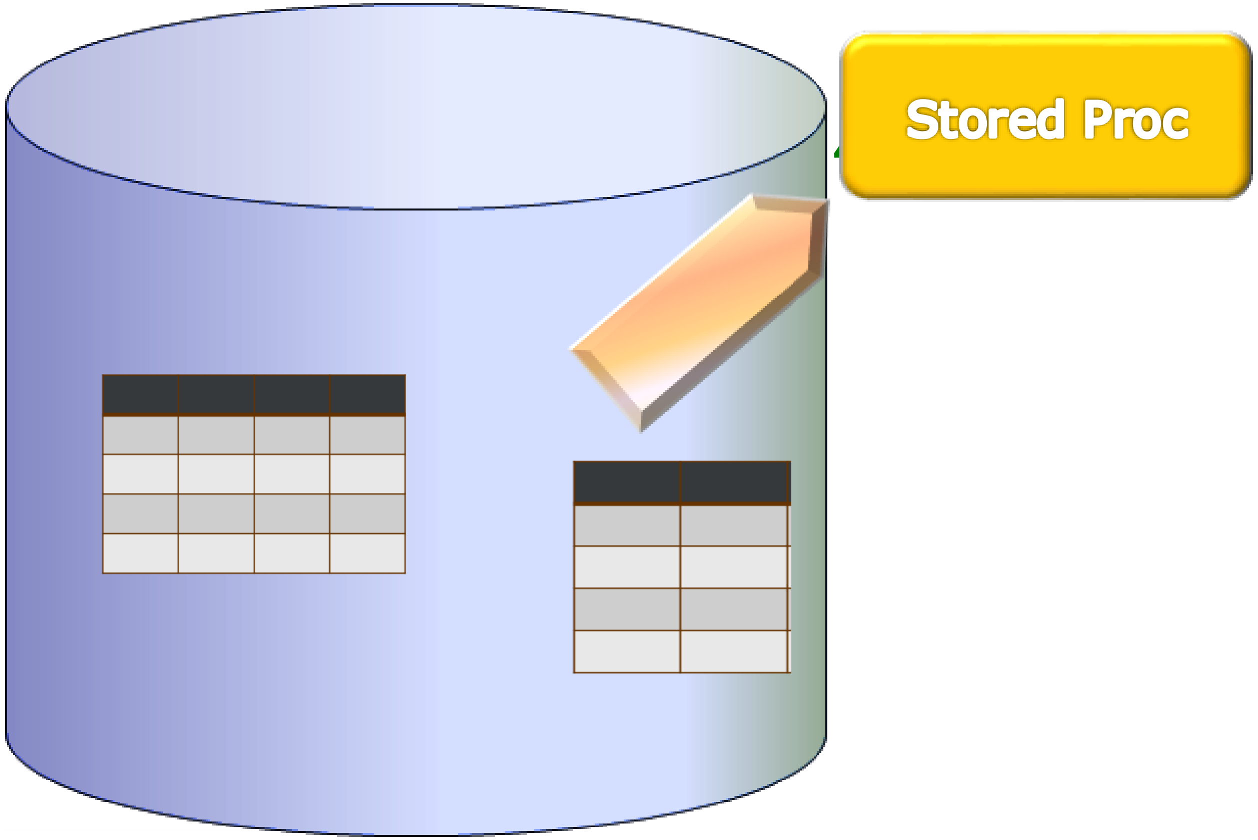
Another case is when you have business rules so complex (exceptions to rules, etc.) that they cannot be checked through declarative constraints. You can abort a transaction in a trigger, and return an error.

❸ Manage data redundancy



A third case is managing some data redundancy. A trigger can write in your back to another table. In the film database, this is done for titles: words are automatically isolated and added to MOVIE_TITLE_FT_INDEX whenever you add a row to MOVIES or ALT_TITLES (not for Chinese titles)

```
insert into movies(title, country, year_released)
values ( 'Monty Python and the Holy Grail', 'gb', 1975   )
```



Trigger Activation

When are triggers fired? "During the change" is not a proper answer. In fact, depending on what the trigger is designed to achieve, it may be fired by various events and at various possible precise moments.

films_francais

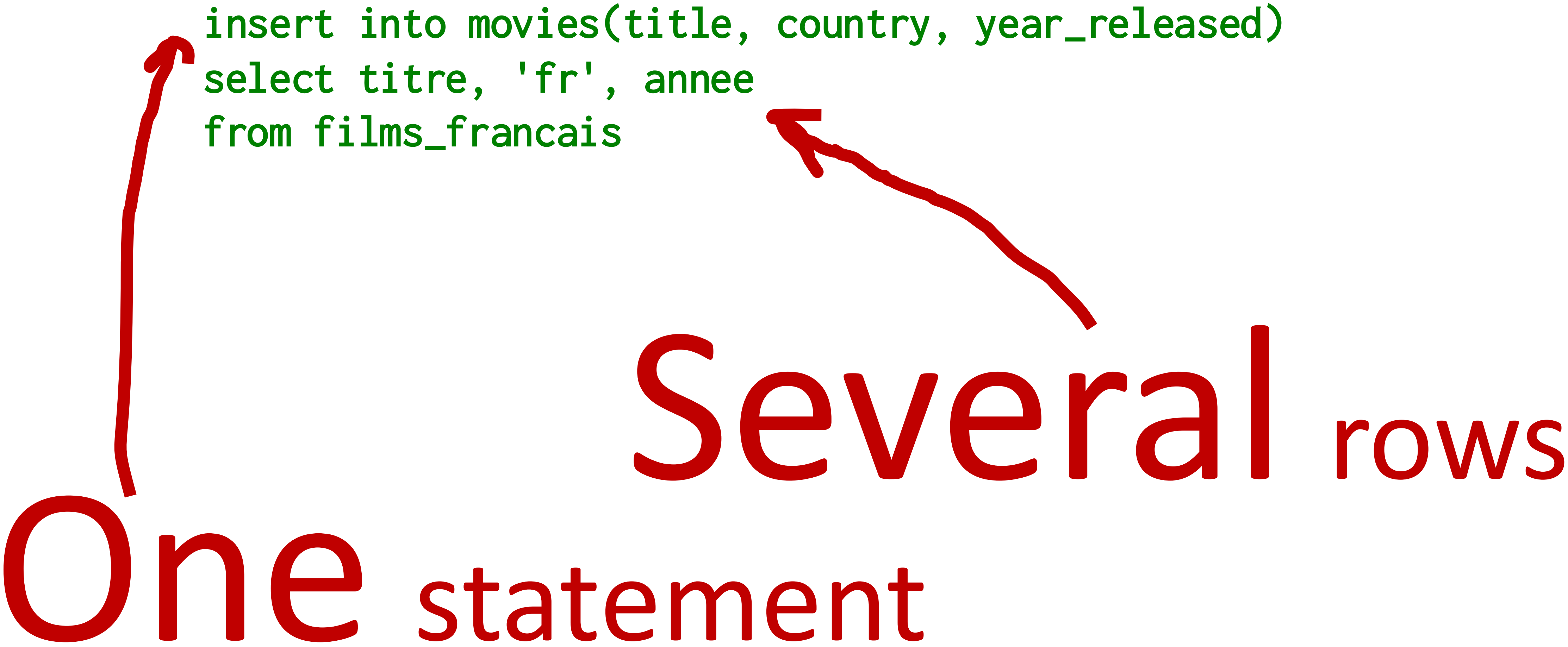
titre

annee

title

year

Let's say that we have uploaded from an external file and into a table called FILMS_FRANCAIS (film is film in French) storing only two columns, title and year.



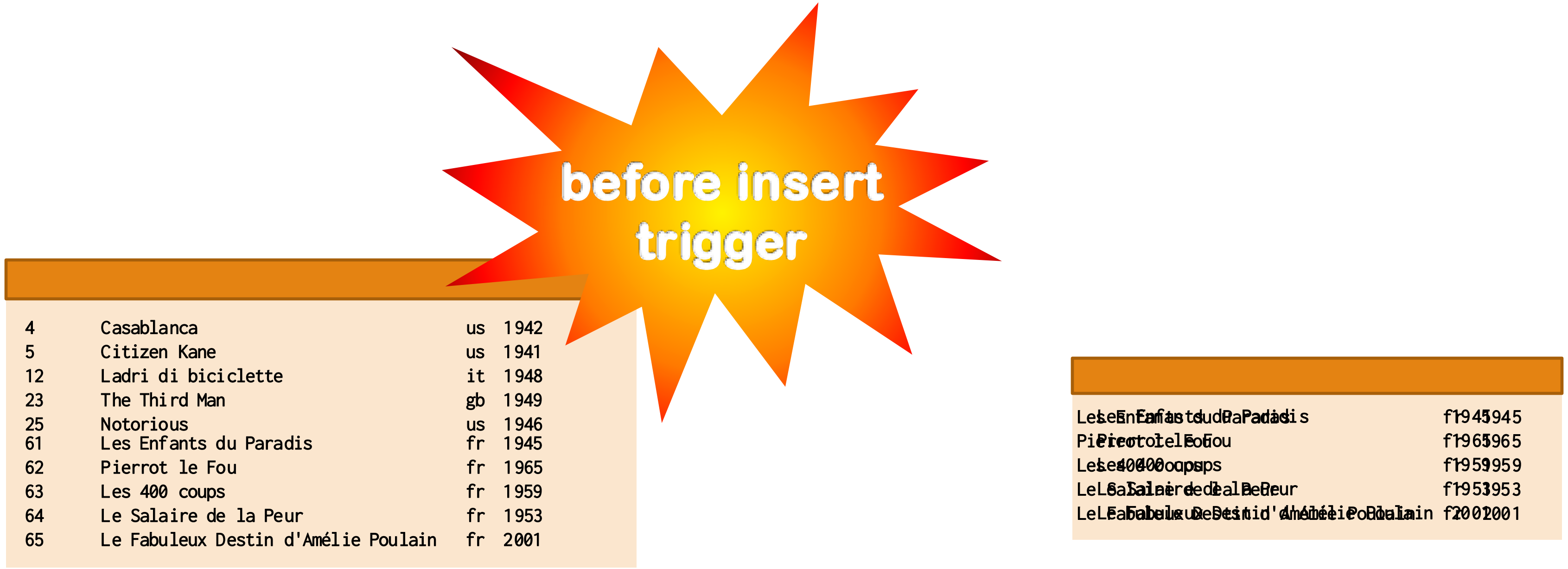
```
insert into movies(title, country, year_released)
select titre, 'fr', annee
from films_francais
```

Several rows

One statement

If we use an INSERT ... SELECT ... statement, we have ONE statement that inserts SEVERAL rows. If we activate a procedure, what will happen? Some DBMS products give you a choice.

```
insert into movies(title, country, year_released)
select titre, 'fr', annee
from films_francais
```



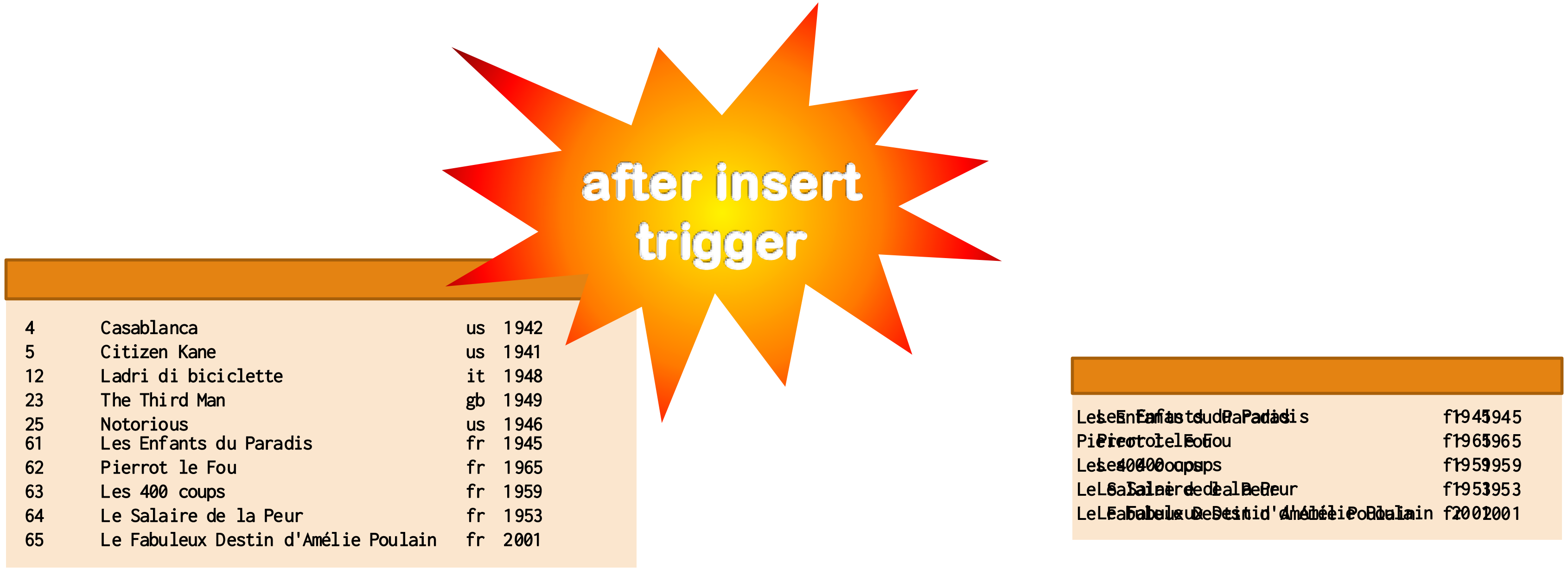
before insert trigger

4	Casablanca	us	1942
5	Citizen Kane	us	1941
12	Ladri di biciclette	it	1948
23	The Third Man	gb	1949
25	Notorious	us	1946
61	Les Enfants du Paradis	fr	1945
62	Pierrot le Fou	fr	1965
63	Les 400 coups	fr	1959
64	Le Salaire de la Peur	fr	1953
65	Le Fabuleux Destin d'Amélie Poulain	fr	2001

Les Enfants du Paradis	fr	1945
Pierrot le Fou	fr	1965
Les 400 coups	fr	1959
Le Salaire de la Peur	fr	1953
Le Fabuleux Destin d'Amélie Poulain	fr	2001

One thing you can sometimes do is fire the procedure only once for the statement, either BEFORE the first row is inserted, or AFTER the last row is inserted.

```
insert into movies(title, country, year_released)
select titre, 'fr', annee
from films_francais
```

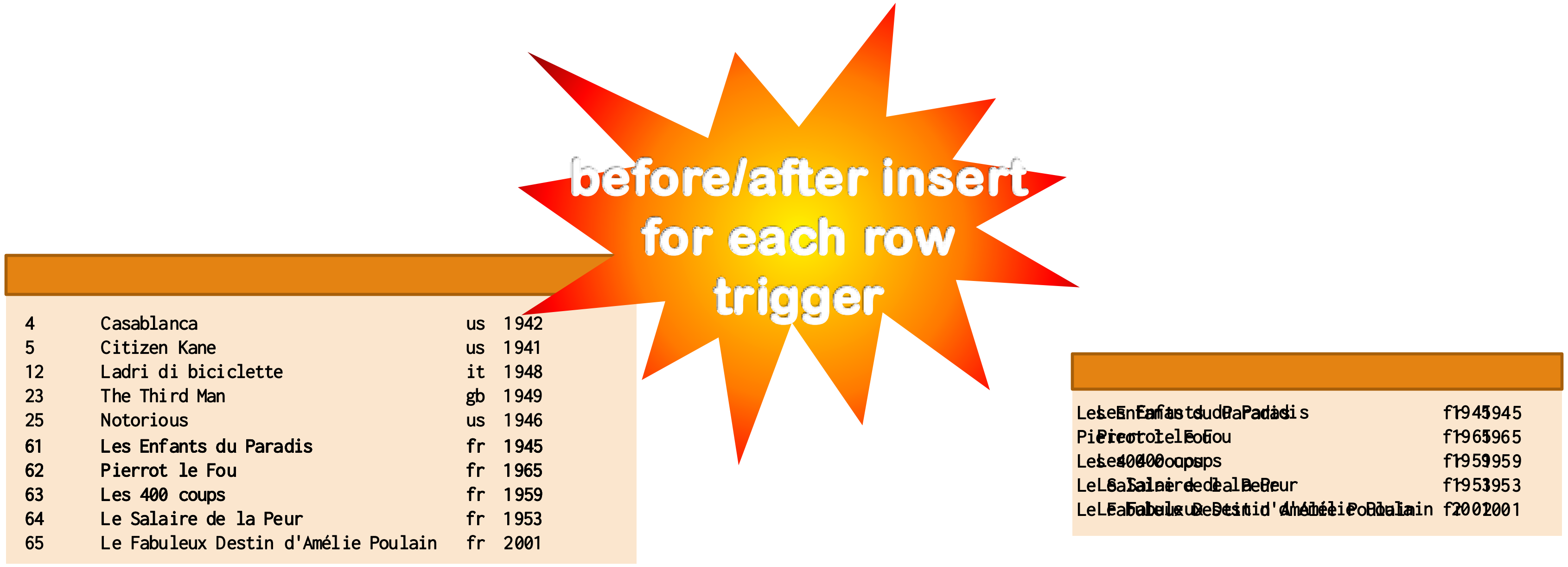


4	Casablanca	us	1942
5	Citizen Kane	us	1941
12	Ladri di biciclette	it	1948
23	The Third Man	gb	1949
25	Notorious	us	1946
61	Les Enfants du Paradis	fr	1945
62	Pierrot le Fou	fr	1965
63	Les 400 coups	fr	1959
64	Le Salaire de la Peur	fr	1953
65	Le Fabuleux Destin d'Amélie Poulain	fr	2001

Les Enfants du Paradis	fr	1945
Pierrot le Fou	fr	1965
Les 400 coups	fr	1959
Le Salaire de la Peur	fr	1953
Le Fabuleux Destin d'Amélie Poulain	fr	2001

OR (and it's sometimes the only option) you can call the procedure before or after you insert EACH row, in which case it will be executed a far greater number of times.

```
insert into movies(title, country, year_released)
select titre, 'fr', annee
from films_francais
```



```
4      Casablanca                       us  1942
5      Citizen Kane                     us  1941
12     Ladri di biciclette              it  1948
23     The Third Man                    gb  1949
25     Notorious                        us  1946
61     Les Enfants du Paradis           fr  1945
62     Pierrot le Fou                   fr  1965
63     Les 400 coups                    fr  1959
64     Le Salaire de la Peur            fr  1953
65     Le Fabuleux Destin d'Amélie Poulain  fr  2001
```

```
Les Enfants du Paradis              fr 1945
Pierrot le Fou                      fr 1965
Les 400 coups                       fr 1959
Le Salaire de la Peur               fr 1953
Le Fabuleux Destin d'Amélie Poulain fr 2001
```

Same thing after each row ...

Time

PostgreSQL

ORACLE

IBM DB2

```
before statement
   before each row
   after each row
after statement
```

old

new

MySQL

SQLite

```
before each row

after each row
```

old

new

Microsoft SQL Server

```
after statement
```

deleted

inserted

IBM DB2

old table

new table

Options vary with DBMS products. Virtual rows or tables give you access to before change/after change values.

Time + Event

insert
update
delete

The other important parameter is WHAT fires the trigger. You don't need to fire a trigger that changes the case when you delete a row.

Several possible triggers

Microsoft SQL Server

PostgreSQL

ORACLE

IBM DB2

```
insert
update
delete
```

Several possible events can fire one trigger

Microsoft SQL Server

PostgreSQL

ORACLE

IBM DB2

```
create trigger trigger_name
before  insert or update or delete
on table_name
for each  row
as
begin
  ...
end
```

Some products let you have several different events that fire the same trigger (timing must be identical)

PostgreSQL

ORACLE

IBM DB2

```
create trigger trigger_name
before delete
on table_name
for each row
as
begin
  ...
end
```

MySQL

SQLite

Other products allow only one trigger per event/timing, and one event per trigger.

```
create trigger trigger_name
on table_name
after insert, update, delete
as
begin
 ...
end
```

SQL Server is a bit special. Triggers are always after the statement, and syntax is different from other products. But several events can fire one trigger.

❶ Modify input on the fly

```
before insert / update
for each row
```

Microsoft® SQL Server™ modify by joining on `inserted`

As I have told you, which trigger you use depends on what you want to do. To modify data on the fly, the trigger must operate on each row, and be fired BEFORE the value is inserted (SQL Server forces you to "fix" things after the row was inserted)

① Modify input on the fly

② Check complex rules

```
before insert / update / delete
for each row
```

Microsoft® SQL Server®

check by joining on `inserted` and `deleted`.
Roll back if something wrong.

Similar story with complex rules. SQL Server is the only product that allows a rollback in a trigger.

① Modify input on the fly

② Check complex rules

③ **Manage data redundancy**

```
after insert / update / delete
for each row
```

Microsoft SQL Server `deleted/inserted`

Data redundancy is only handled when the triggering event was successful, therefore AFTER.

9.3 Auditing

Shiqi Yu 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult’s slides

Auditing

One good example of managing some data redundancy is keeping an audit trail. It won't do anything for people who steal data (remember that SELECT cannot fire a trigger – although with the big products you can trace all queries), but it may be useful for checking people who modify data that they aren't supposed to modify. We'll do it with PostgreSQL.

This is what an audit table might look like. We'll store one row per changed column in the PEOPLE table.

```
create table people_audit(auditid         serial,
                          peopleid        int not null,
                          type_of_change  char(1),
                          column_name     varchar,
                          old_value       varchar,
                          new_value       varchar,
                          changed_by      varchar,
                          time_changed    datetime);
```

Multiple ways to do it ...

Another option might be to have one big string storing all the changes in XML or JSON format for instance.

```
create or replace function people_audit_fn()
returns trigger
as $$
begin
  if tg_op = 'UPDATE'
  then
    insert into people_audit(peopleid,
                          type_of_change,
                          column_name,
                          old_value,
                          new_value,
                          changed_by,
                          time_changed);
    select peopleid, 'U', column_name, old_value, new_value,
              current_user || '@'
                 || coalesce(cast(inet_client_addr() as varchar, 'localhost'),
           current_timestamp
```

With PostgreSQL (only) you need to create a special function that returns a trigger.

TG_OP is a system variable that says which operation fired the trigger (with other products you might say "when updating then")

```
from (select old.peopleid,
             'first_name' column_name,
             old.first_name old_value,
             new.first_name new_value
         where coalesce(old.first_name, '*')
                            <> coalesce(new.first_name, '*')
    union all
      select old.peopleid,
             'surname' column_name,
             old.surname old_value,
             new.surname new_value
         where old.surname <> new.surname
    union all
      select old.peopleid,
             'born' column_name,
             cast(old.born as varchar) old_value,
             cast(new.born as varchar) new_value
         where old.born <> new.born
    union all
      select old.peopleid,
             'died' column_name,
             cast(old.died as varchar) old_value,
             cast(new.died as varchar) new_value
          where coalesce(old.died, -1) <> coalesce(new.died, -1)) modified;
```

Painful statement checking column by column if it was changed.

```
create or or replace function people_audit_fn()
returns trigger
as
$$
begin
  if tg_op = 'UPDATE'
  then
    insert into people_audit(...)
    ...
  elsif tg_op = 'INSERT' then
    insert into people_audit(...)
    ...
  else
    insert into people_audit(...)
    ...
  end if;
  return null;
end;
$$ language plpgsql;
```

Rinse, repeat

It's easier for inserts and for deletes because every not null column should be recorded. For inserts values are in the NEW dummy row, and for deletes in the OLD one.

An AFTER/FOR EACH ROW trigger doesn't need to return anything; a BEFORE/FOR EACH ROW trigger must return a (usually modified) "row variable" which will be what the SQL engine will use for the operation

Once the function is ready you call it in the trigger. With other products you could have the whole code in the trigger body (between **begin … end**), or call there a regular stored procedure.

```
create trigger people_trg
after insert or update or delete on people
for each row
execute procedure people_audit_fn();
```

not "function" …

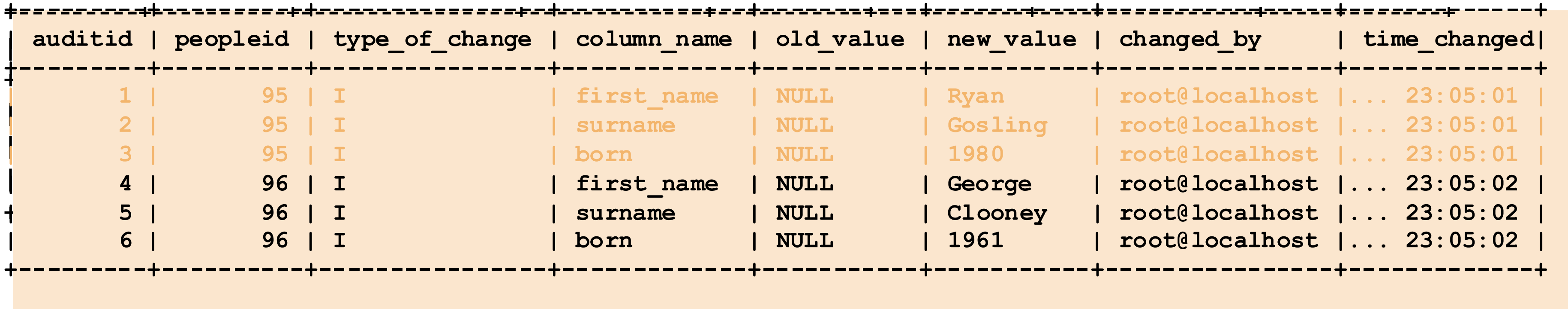
```
insert into people(first_name, surname, born)
values('Ryan', 'Gosling', 1980);
```

With the trigger, every new, not null value will be recorded.

people_audit

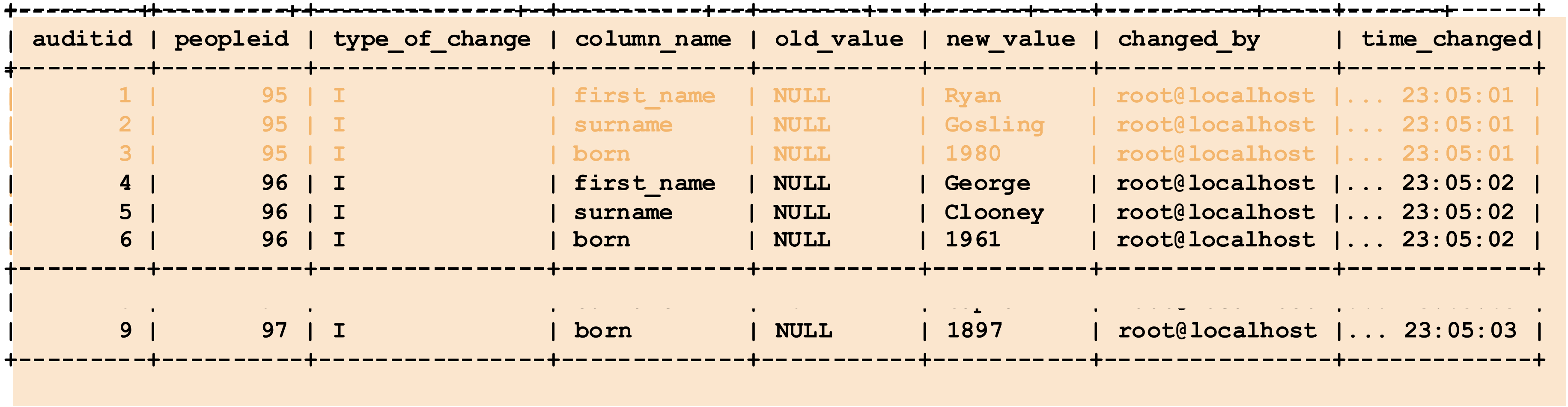
```
+---------+----------+---------------+-------------+-----------+-----------+----------------+-------------+
| auditid | peopleid | type_of_change | column_name | old_value | new_value | changed_by     | time_changed|
+---------+----------+---------------+-------------+-----------+-----------+----------------+-------------+
|       1 |       95 | I              | first_name  | NULL      | Ryan      | root@localhost |... 23:05:01 |
|       2 |       95 | I              | surname     | NULL      | Gosling   | root@localhost |... 23:05:01 |
|       3 |       95 | I              | born        | NULL      | 1980      | root@localhost |... 23:05:01 |
+---------+----------+---------------+-------------+-----------+-----------+----------------+-------------+
insert into people(first_name, surname, born)
values('George', 'Clooney', 1961);
```

people_audit



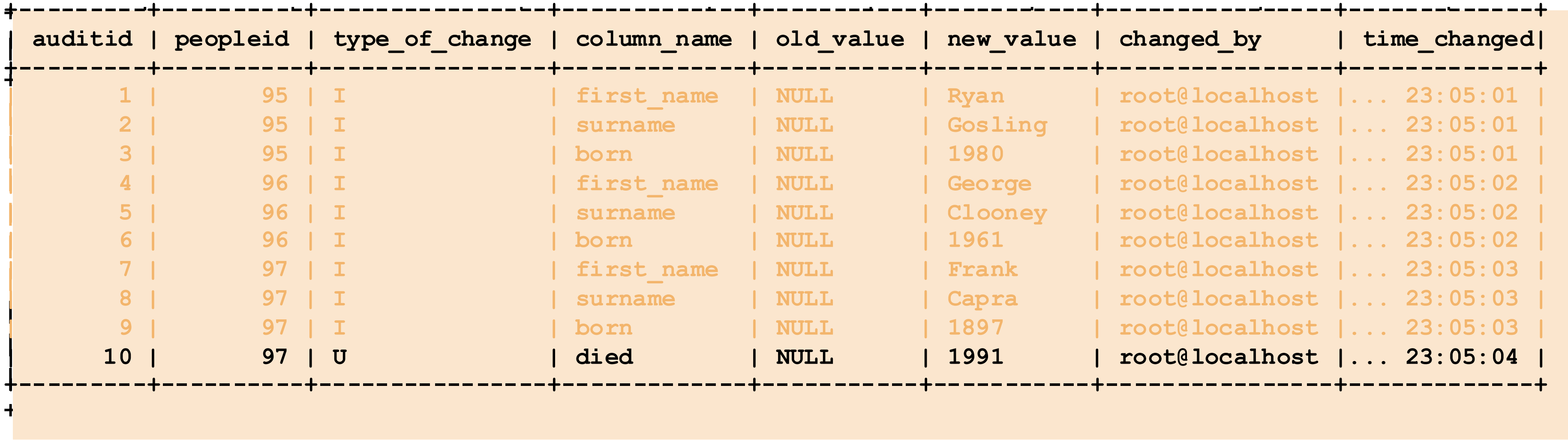
```
+---------+----------+----------------+-------------+-----------+-----------+----------------+-------------+
| auditid | peopleid | type_of_change | column_name | old_value | new_value | changed_by     | time_changed|
+---------+----------+----------------+-------------+-----------+-----------+----------------+-------------+
|       1 |       95 | I              | first_name  | NULL      | Ryan      | root@localhost |... 23:05:01 |
|       2 |       95 | I              | surname     | NULL      | Gosling   | root@localhost |... 23:05:01 |
|       3 |       95 | I              | born        | NULL      | 1980      | root@localhost |... 23:05:01 |
|       4 |       96 | I              | first_name  | NULL      | George    | root@localhost |... 23:05:02 |
|       5 |       96 | I              | surname     | NULL      | Clooney   | root@localhost |... 23:05:02 |
|       6 |       96 | I              | born        | NULL      | 1961      | root@localhost |... 23:05:02 |
+---------+----------+----------------+-------------+-----------+-----------+----------------+-------------+
insert into people(first_name, surname, born)
values('Frank', 'Capra', 1897);
```

people_audit



```
+---------+----------+----------------+-------------+-----------+-----------+-----------------+-------------+
| auditid | peopleid | type_of_change | column_name | old_value | new_value | changed_by      | time_changed|
+---------+----------+----------------+-------------+-----------+-----------+-----------------+-------------+
|       1 |       95 | I              | first_name  | NULL      | Ryan      | root@localhost |... 23:05:01 |
|       2 |       95 | I              | surname     | NULL      | Gosling   | root@localhost |... 23:05:01 |
|       3 |       95 | I              | born        | NULL      | 1980      | root@localhost |... 23:05:01 |
|       4 |       96 | I              | first_name  | NULL      | George    | root@localhost |... 23:05:02 |
|       5 |       96 | I              | surname     | NULL      | Clooney   | root@localhost |... 23:05:02 |
|       6 |       96 | I              | born        | NULL      | 1961      | root@localhost |... 23:05:02 |
+---------+----------+----------------+-------------+-----------+-----------+-----------------+-------------+
|
|       9 |       97 | I              | born        | NULL      | 1897      | root@localhost |... 23:05:03 |
+---------+----------+----------------+-------------+-----------+-----------+-----------------+-------------+
update people
set died = 1991
where first_name = 'Frank'
  and surname = 'Capra';
```

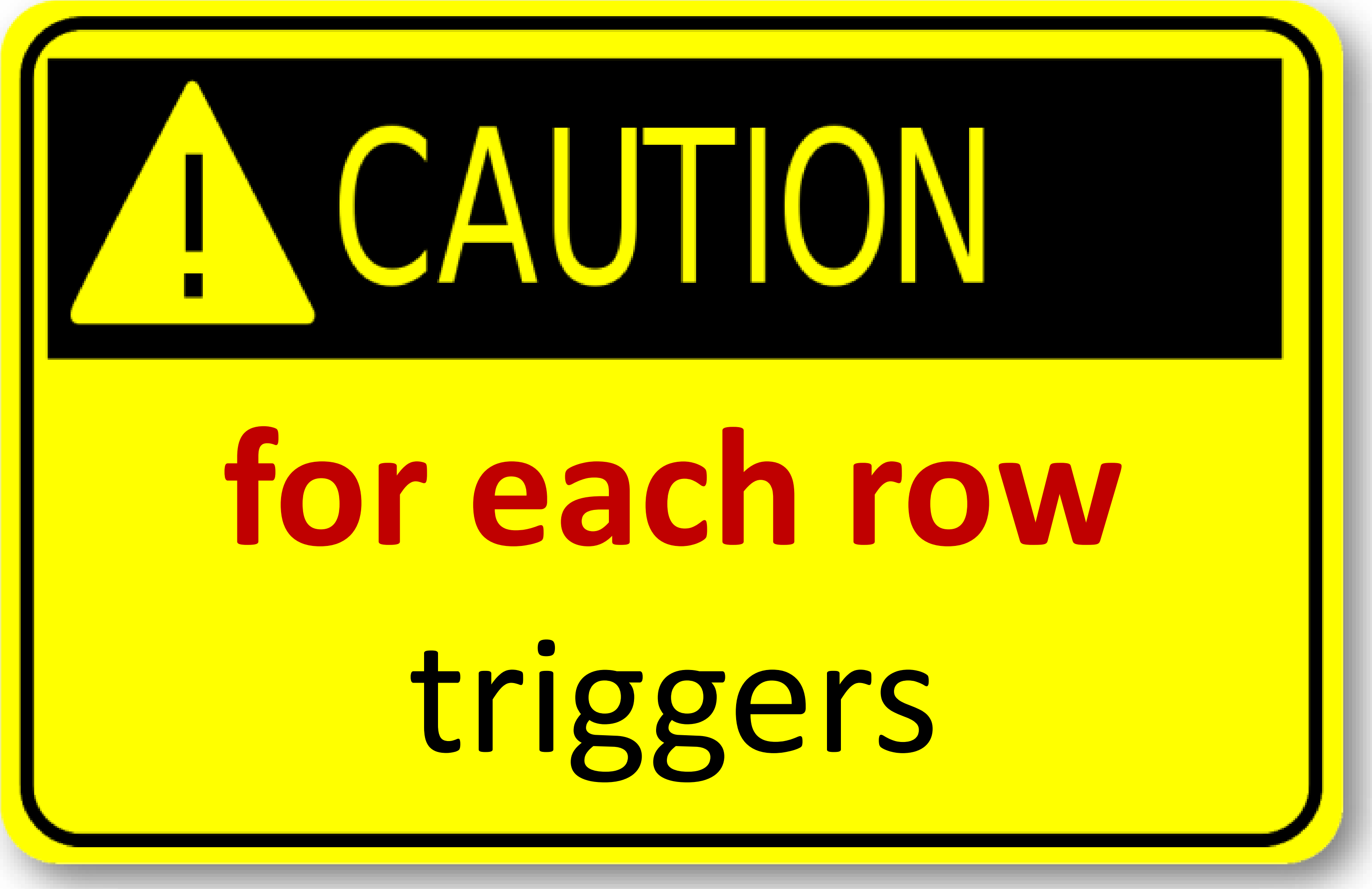
people_audit



```
+---------+----------+----------------+-------------+-----------+-----------+----------------+-------------+
| auditid | peopleid | type_of_change | column_name | old_value | new_value | changed_by     | time_changed|
+---------+----------+----------------+-------------+-----------+-----------+----------------+-------------+
|       1 |       95 | I              | first_name  | NULL      | Ryan      | root@localhost |... 23:05:01 |
|       2 |       95 | I              | surname     | NULL      | Gosling   | root@localhost |... 23:05:01 |
|       3 |       95 | I              | born        | NULL      | 1980      | root@localhost |... 23:05:01 |
|       4 |       96 | I              | first_name  | NULL      | George    | root@localhost |... 23:05:02 |
|       5 |       96 | I              | surname     | NULL      | Clooney   | root@localhost |... 23:05:02 |
|       6 |       96 | I              | born        | NULL      | 1961      | root@localhost |... 23:05:02 |
|       7 |       97 | I              | first_name  | NULL      | Frank     | root@localhost |... 23:05:03 |
|       8 |       97 | I              | surname     | NULL      | Capra     | root@localhost |... 23:05:03 |
|       9 |       97 | I              | born        | NULL      | 1897      | root@localhost |... 23:05:03 |
|      10 |       97 | U              | died        | NULL      | 1991      | root@localhost |... 23:05:04 |
+---------+----------+----------------+-------------+-----------+-----------+----------------+-------------+
delete from people
where first_name = 'Ryan'
  and surname = 'Gosling';
```

people_audit

```
+---------+----------+----------------+-------------+-----------+-----------+----------------+-------------+
| auditid | peopleid | type_of_change | column_name | old_value | new_value | changed_by     | time_changed|
+---------+----------+----------------+-------------+-----------+-----------+----------------+-------------+
|       1 |       95 | I              | first_name  | NULL      | Ryan      | root@localhost |... 23:05:01 |
|       2 |       95 | I              | surname     | NULL      | Gosling   | root@localhost |... 23:05:01 |
|       3 |       95 | I              | born        | NULL      | 1980      | root@localhost |... 23:05:01 |
|       4 |       96 | I              | first_name  | NULL      | George    | root@localhost |... 23:05:02 |
|       5 |       96 | I              | surname     | NULL      | Clooney   | root@localhost |... 23:05:02 |
|       6 |       96 | I              | born        | NULL      | 1961      | root@localhost |... 23:05:02 |
|       7 |       97 | I              | first_name  | NULL      | Frank     | root@localhost |... 23:05:03 |
|       8 |       97 | I              | surname     | NULL      | Capra     | root@localhost |... 23:05:03 |
|       9 |       97 | I              | born        | NULL      | 1897      | root@localhost |... 23:05:03 |
|      10 |       97 | U              | died        | NULL      | 1991      | root@localhost |... 23:05:04 |
|      11 |       95 | D              | first_name  | Ryan      | NULL      | root@localhost |... 23:05:05 |
|      12 |       95 | D              | surname     | Gosling   | NULL      | root@localhost |... 23:05:05 |
|      13 |       95 | D              | born        | 1980      | NULL      | root@localhost |... 23:05:05 |
+---------+----------+----------------+-------------+-----------+-----------+----------------+-------------+
```



Beware of FOR EACH ROW triggers, you cannot do anything in them.

```
SQL> create table test(id int, label varchar(20), unique(id));

Table created.

SQL> insert into test(id, label) values(1, 'This is line 1');

1 row created.

SQL> insert into test(id, label) values(2, 'This is line 2');

1 row created.

SQL> select * from test;

    ID LABEL
---------- --------------------
     1 This is line 1
     2 This is line 2

SQL>
```

Take note!

```
SQL> update test set id = case id when 1 then 2 else 1 end;

2 rows updated.

SQL> select * from test;

   ID LABEL
---------- --------------------
    2 This is line 1
    1 This is line 2

SQL>
```

If I switch the values between the two columns, it works (same behaviour with all DBMS products except PostgreSQL)

Value of id in the other row when you update one row?

Constraint?

STABLE

CONSISTENT

STATE

What happens is that consistency and constraints are checked AFTER the update, not DURING. During the update, the state is undefined.

UPDATE

STABLE

CONSISTENT

STATE

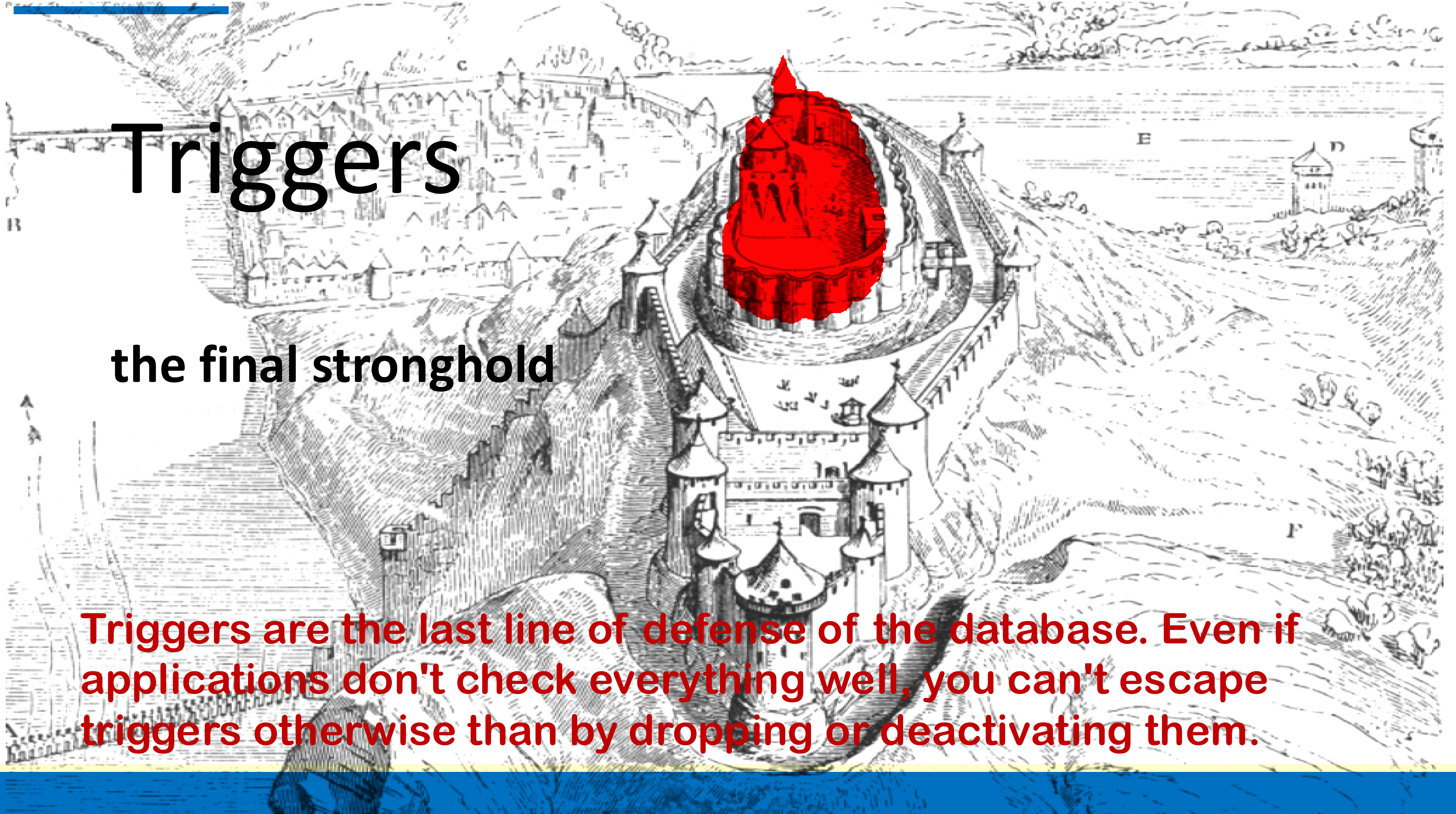
Flickr: Alex Proimos

DON'T

look at
other rows
of the
modified table
in
for each row
triggers

Triggers

the final stronghold



Triggers are the last line of defense of the database. Even if applications don't check everything well, you can't escape triggers otherwise than by dropping or deactivating them.

Triggers = complexity

if you can,

AVOID

triggers

This being said, they add a lot of complexity, a simple operation may behave weirdly because of what a poorly written trigger does, and triggers are pretty much below the radar. Knowing whether a trigger is active or not requires special checks.

Additionally, they are often used to "fix" issues that should not have existed in the first place and often result of a poor database design.

if possible ...

Don't use triggers to fix design issues

Use stored procedures preferably to triggers

However, if users can access the database otherwise than through your programs ...

Use triggers if there are multiple access points

Use functions to reuse complex expressions

Use functions to reuse complex expressions

Don't query the database in functions

Use functions to reuse complex expressions

Don't query the database in functions

Use procedures for business operations

Use functions to reuse complex expressions

Don't query the database in functions

Use procedures for business operations

Procedures aren't where to be heavily procedural

Use functions to reuse complex expressions

Don't query the database in functions

Use procedures for business operations

Procedures aren't where to be heavily procedural

Triggers: the last line of defense.
Only when you can't do otherwise.