CS307

Database Principles

Stéphane Faroult
faroult@sustc.edu.cn

Liu Zijian liuzijian47@163.com

Procedures

Business processes

Avoid cursors

Use insert ... select ... and complex updates

This is what I think you should primarily remember about procedures. They should pack efficient statements together, not degenerate into complicated logic.

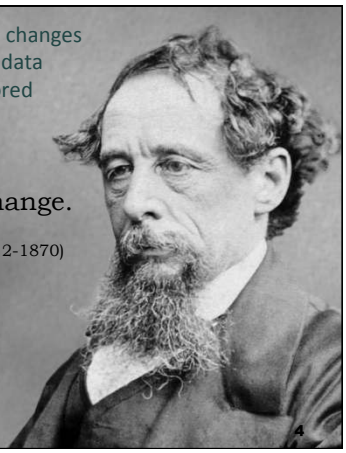
Triggers

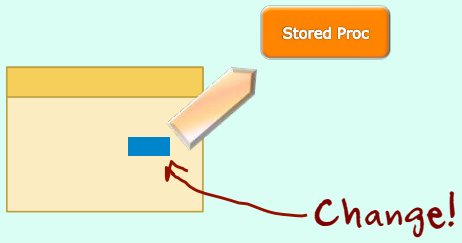
Stored procedures are usually explicitly called. There is also a mechanism for executing stored procedures automatically: triggers.

Triggers are fired by data changes (never by a SELECT). Any data change can activate a stored procedure.

Change begets change.

Charles Dickens (1812-1870)

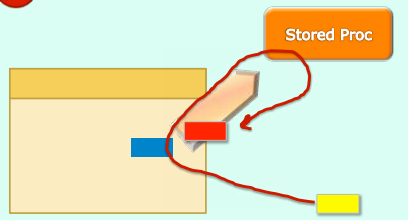




PURPOSE

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 in fixing things which are badly done by a program for which you haven't the source code.

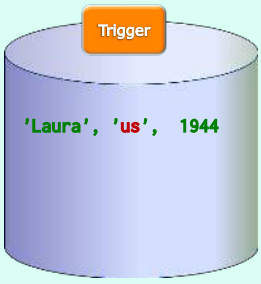
1 Modify input on the fly



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 (bought) data entry program doesn't enforce it. A trigger can force the case.

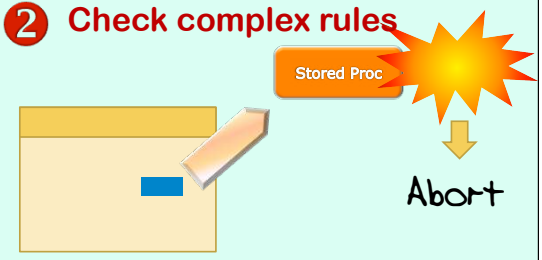
Note!

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



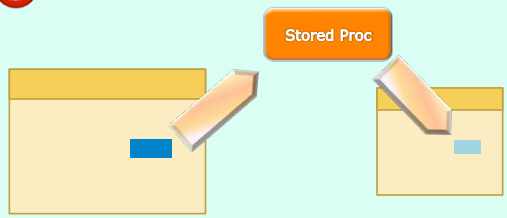
Much better to have the transformation in input programs ....

2 Check complex rules



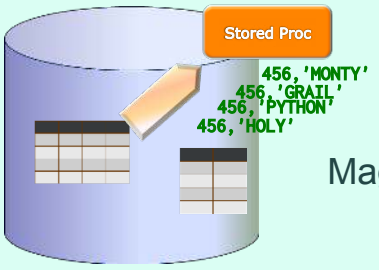
Another case is when you have business rules so complex (exceptions to rules, etc.) that they cannot be checked through declarative constraints. You can abort a transaction in a trigger, and return an error.

3 Manage data redundancy



A third case is managing some data redundancy (which means some duplication of data). 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_INDEX2 whenever you add a row to MOVIES or ALT_TITLES.

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



Magic.

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

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.

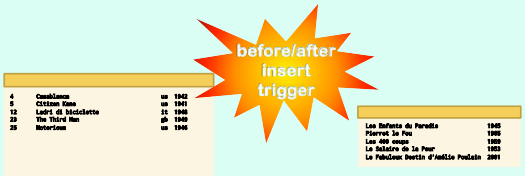
```
insert into movies(title, country, year_released)
select titre, 'fr', annee
from films_francais
```

Several rows

One statement

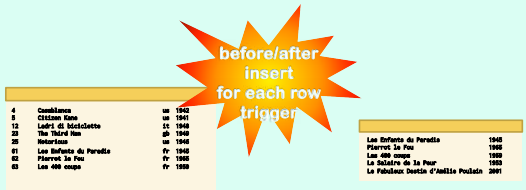
If we use an INSERT ... SELECT ... statement, we have ONE statement that inserts SEVERAL rows. If we activate a procedure, what will happen? Some DBMS products give you a choice.

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



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
```



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.

Time

PostgreSQL, ORACLE, IBM DB2	MySQL, SQLite	SQL Server
before statement before each row after each row after statement	before each row after each row	after statement
old new	old new	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

Several possible events can fire one trigger

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
```

PostgreSQL, ORACLE, IBM DB2

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

```
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
```

SQLServer

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.

1 Modify input on the fly

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

SQLServer 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
```

SQLServer 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
```

SQLServer **deleted/inserted**

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

Auditing

One good example of managing some data redundancy is keeping an audit trail. It won't do anything for people who steal data, but it may be useful for checking people who modify data that they aren't supposed to modify (at least not in a particular way). 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(30),
                          old_value        varchar(250),
                          new_value        varchar(250),
                          changed_by       varchar(100),
                          time_changed     timestamp);
```

All columns except old/new_value could be made NOT NULL

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.

We collect information about the user running the update

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.

Only changes are recorded

Can be null

```
create 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.

Notice that the initial "returns trigger" is completely dummy. We can return anything, null is OK.

Once the function is ready you call it in the trigger. With other products you could have the whole code in the trigger body, or call 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);

insert into people(first_name, surname, born)
values('Frank', 'Capra', 1897);

update people
set died = 1991
where first_name = 'Frank'
  and surname = 'Capra';

delete from people
where first_name = 'Ryan'
  and surname = 'Gosling';
```

We may perform many changes, and even delete the actor we had first inserted.

Every single change to the table will be recorded, and even if in PEOPLE Ryan Gosling was deleted, we'll know when he was added and deleted, and by whom.

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!

I am going to illustrate (here with Oracle) something that might look strange. I have a uniqueness constraint.

```
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 (except with PostgreSQL)

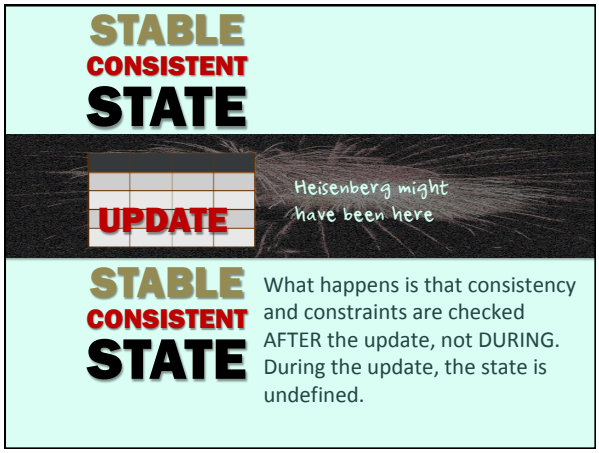
Value of id in the other row when you update one row?

Constraint?

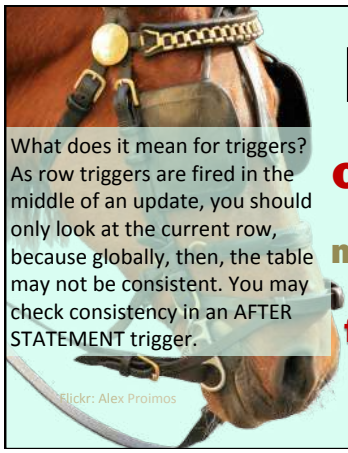
Except with PostgreSQL, unless ...

```
create table test
          (id int,
           label varchar(20),
           unique(id) deferrable initially deferred);
```

You need to do something a bit special with PostgreSQL to make it behave like other DBMS products (its default behavior isn't standard)



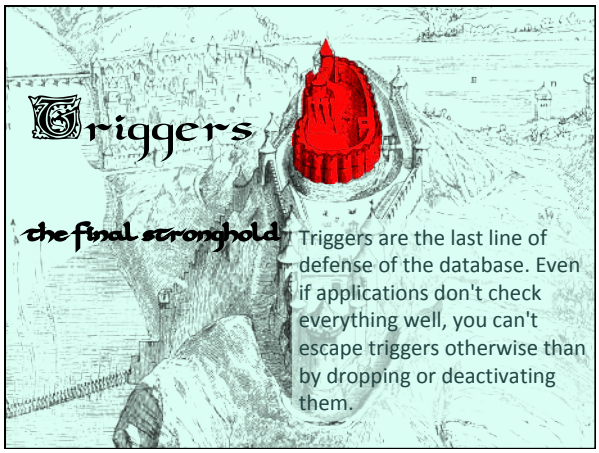
What happens is that consistency and constraints are checked AFTER the update, not DURING. During the update, the state is undefined.



What does it mean for triggers? As row triggers are fired in the middle of an update, you should only look at the current row, because globally, then, the table may not be consistent. You may check consistency in an AFTER STATEMENT trigger.

Flickr: Alex Proimos

DON'T look at other rows of the modified table in for each row triggers



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

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.

if you can,

AVOID

triggers

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**

Problem

SQL

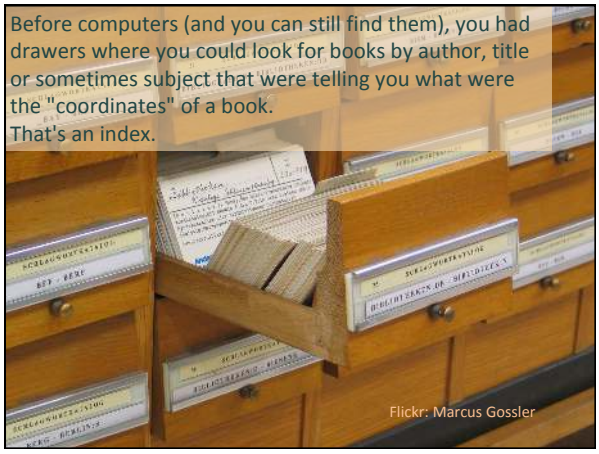
Solution

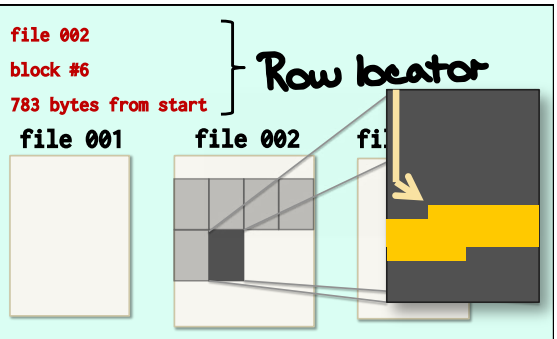
What we have seen so far is how, when you are given a question about data, you can use SQL to get the answer, and we have focused on getting the RIGHT answer.
The presence of nulls and duplicates can very well give an answer that looks right and isn't.

And if I fully agree with Wyatt, he would probably have agreed with me that being too slow isn't too good either. We need to be accurate AND fast.

SPEEDING IT UP

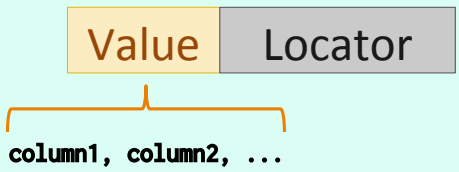
So let's see what might help find data faster in a table (imagine that you are looking for people by name). It's like with books. If you have few books at home and are looking for one, you can check them one by one. In a bookshop, books need to be ordered by theme. In a big library, they need a very strict organization. Moreover, you cannot rely on any physical order, because some people may look for a book by author, and others by title.





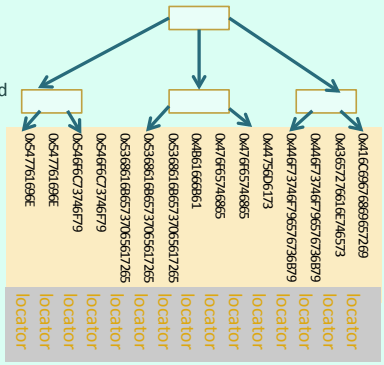
We'll see the structure in more detail later, but a database is made of files, themselves usually organized in equal-sized "pages" (or "blocks"). File, block and offset allow to locate very fast any row.

The whole idea of indexing consists in associating values in one or several columns of a table (we may look by groups of columns, such as FIRST_NAME and SURNAME in the PEOPLE table, and want to index them as a combination) to the locator(s) of the row(s) where they can be found.



We build a sorted list of all the values with their locators.

Remember that indexed values are binary values, that will be important!



To search the list easily, we plug a tree above it.

You create an index by givng it a name and specifying tablename and column(s)

```
create index <index name>
on <table name>(<col1>, ... <coln>)
```

Example:

```
create index countries_cont_idx
on countries(continent)
```

Two columns often queried together can be indexed together; what is indexed is concatenated values (NOT separate values)

```
create index people_surname_born_idx
on people(surname, born)
```

Composite index

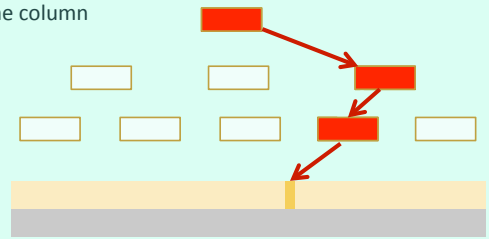
You have actually already created indexes without knowing it: whenever you declare a PRIMARY KEY or UNIQUE constraint, an index is created behind your back.

Primary key → INDEX

Unique constraint → INDEX

It has nothing to do with constraints or the relational theory (with indexes, we are more talking engineering than theory), it's purely practical.

The only way to find quickly in a big table that a supposedly unique value is already recorded is to index the column



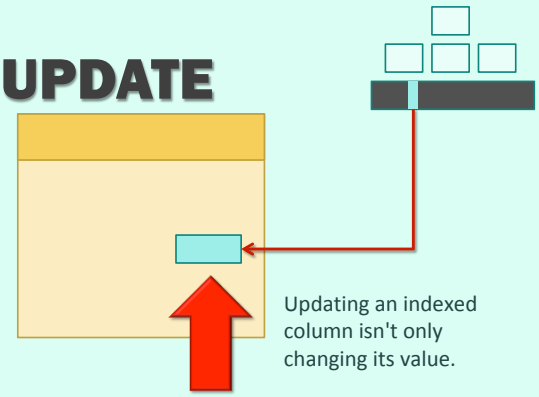
Ooops ...
Already there ...

INSERT DELETE

Then we should certainly index all columns? Why isn't it done by default? In fact, everything isn't rosy: insertion and deletion always require maintaining table AND indexes. Quite a lot of work.

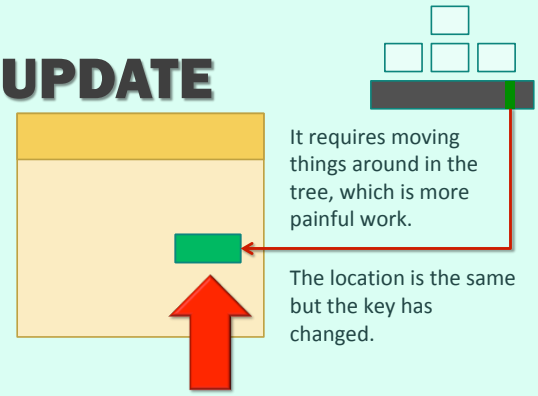
Table
+ Index
+ Index
+ Index
+ Index

UPDATE



Updating an indexed column isn't only changing its value.

UPDATE



It requires moving things around in the tree, which is more painful work.

The location is the same but the key has changed.

Storage

Additionally, indexes use a lot of storage, sometimes more than data! It has a huge impact on operations.

(By "operations" I mean regular activities such as backups)

You can also declare an index to be unique.

```
create unique index <index name>
on <table name>(<col1>, ... <coln>)
```

Enforces unique constraint like a constraint definition

If both are equivalent, then which one should we use?

Unique index

Unique constraint

Constraints

Logic Rules

Constraints, without hesitation. They refer to design.

Indexes

Implementation

However (there is always a "however") there are some rare cases when some uniqueness (such as case-insensitive uniqueness in Oracle in a column in which data is in mixed case) cannot be enforced through declarative constraints but can be with the dirty trick of unique indexes.