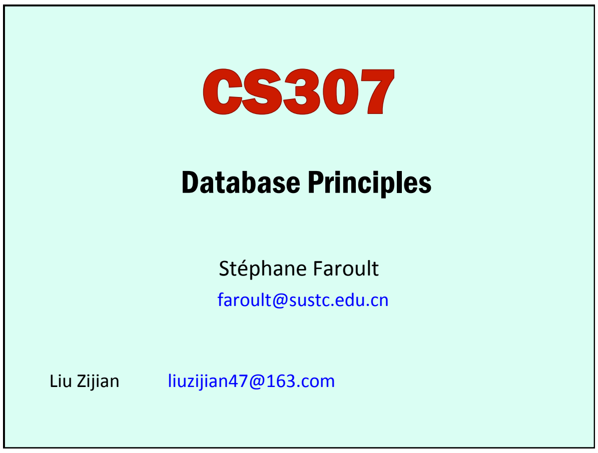
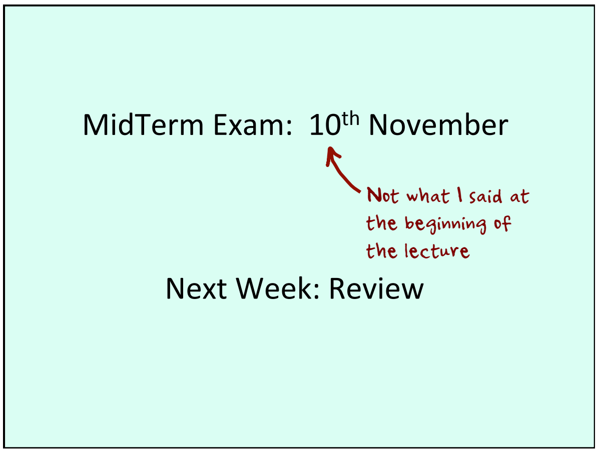


CS307

Database Principles

Stéphane Faroult
faroult@sustc.edu.cn

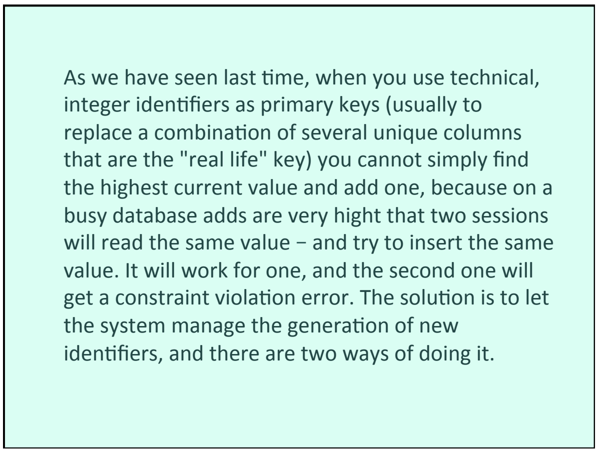
Liu Zijian liuzijian47@163.com



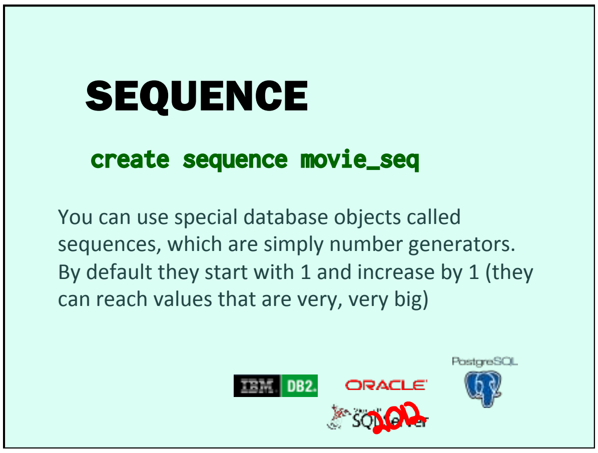
MidTerm Exam: 10th November

Not what I said at the beginning of the lecture

Next Week: Review



As we have seen last time, when you use technical, integer identifiers as primary keys (usually to replace a combination of several unique columns that are the "real life" key) you cannot simply find the highest current value and add one, because on a busy database adds are very hight that two sessions will read the same value – and try to insert the same value. It will work for one, and the second one will get a constraint violation error. The solution is to let the system manage the generation of new identifiers, and there are two ways of doing it.

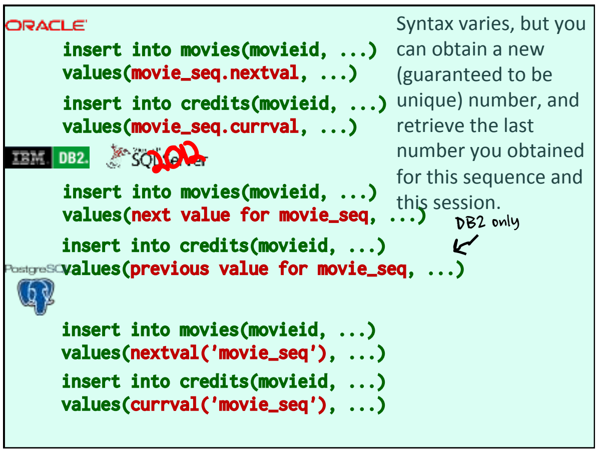


SEQUENCE

```
create sequence movie_seq
```

You can use special database objects called sequences, which are simply number generators. By default they start with 1 and increase by 1 (they can reach values that are very, very big)

IBM DB2
ORACLE
SQL Server
2012
PostgreSQL



ORACLE

```
insert into movies(movieid, ...)
values(movie_seq.nextval, ...)
insert into credits(movieid, ...)
values(movie_seq.currval, ...)
```

IBM DB2 SQL Server 2012

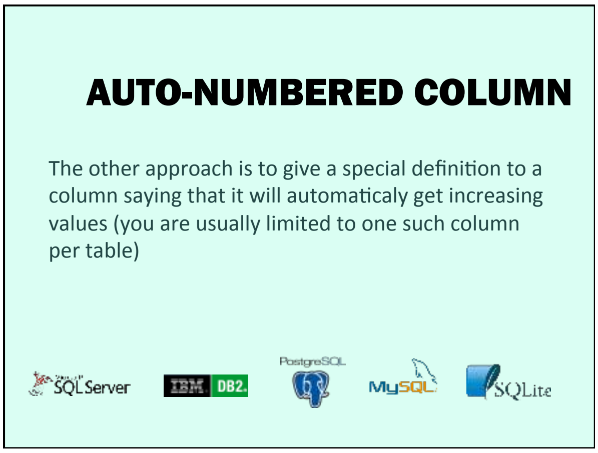
```
insert into movies(movieid, ...)
values(next value for movie_seq, ...)
insert into credits(movieid, ...)
values(previous value for movie_seq, ...)
```

DB2 only

PostgreSQL

```
insert into movies(movieid, ...)
values(nextval('movie_seq'), ...)
insert into credits(movieid, ...)
values(currval('movie_seq'), ...)
```

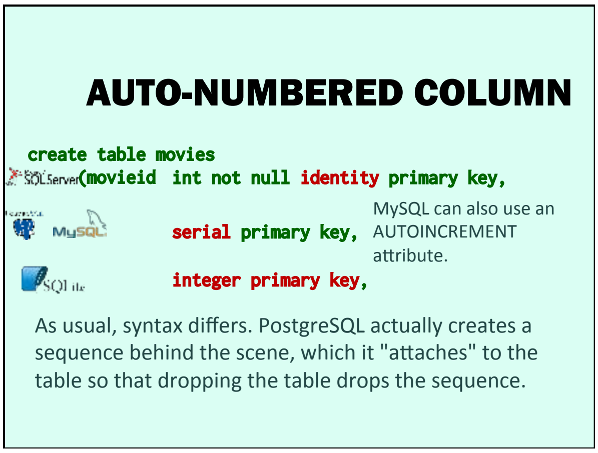
Syntax varies, but you can obtain a new (guaranteed to be unique) number, and retrieve the last number you obtained for this sequence and this session.



AUTO-NUMBERED COLUMN

The other approach is to give a special definition to a column saying that it will automaticaly get increasing values (you are usually limited to one such column per table)

SQL Server IBM DB2 PostgreSQL MySQL SQLite



AUTO-NUMBERED COLUMN

```
create table movies
(movieid  int not null identity primary key,
          serial primary key,
          integer primary key,
```

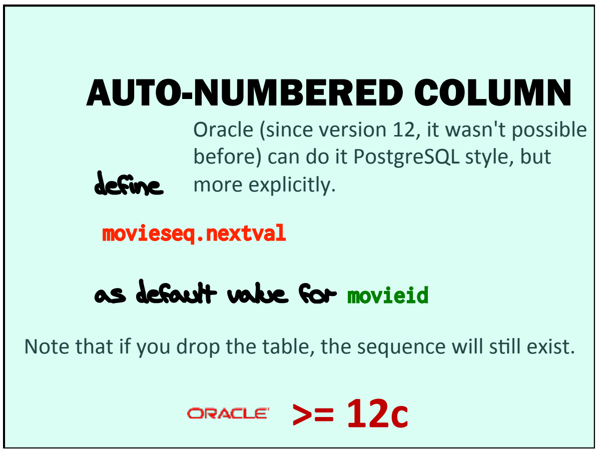
SQL Server: `identity primary key`

PostgreSQL, MySQL: `serial primary key`

MySQL can also use an AUTOINCREMENT attribute.

SQLite: `integer primary key`

As usual, syntax differs. PostgreSQL actually creates a sequence behind the scene, which it "attaches" to the table so that dropping the table drops the sequence.



AUTO-NUMBERED COLUMN

Oracle (since version 12, it wasn't possible before) can do it PostgreSQL style, but more explicitly.

define

`movieseq.nextval`

as default value for `movieid`

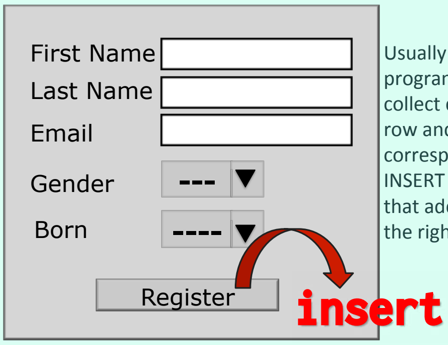
Note that if you drop the table, the sequence will still exist.

ORACLE >= 12c

If you insert a film with an auto-numbered column, you just omit the movieid from the INSERT statement, it will get automatically populated.

```
insert into movies(title, ...)
values('Some Movie Title', ...)
```

To retrieve the last value generated in your session, you use a special variable such as `@@identity` with SQL Server, or functions with other products (eg. `lastval()` with PostgreSQL or `last_insert_id()` with MySQL). This last value is often "across all tables", although sometimes you can retrieve the last value for a specific table.



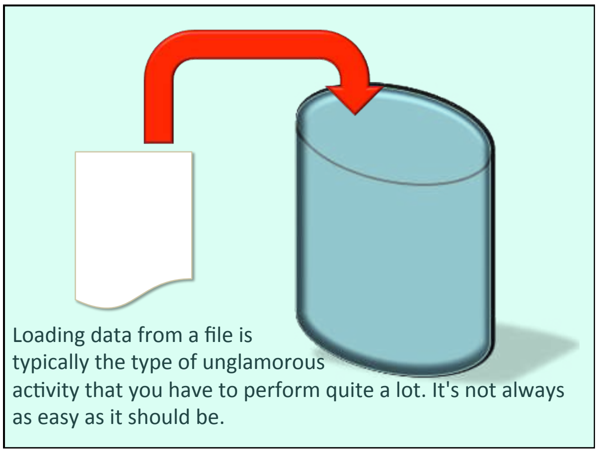
Usually in interactive programs, interfaces collect data for one row and issue the corresponding INSERT statement that adds the row to the right table.

However, it's extremely common that you want to upload data in bulk, either to populate a database initially or because data is exchanged between different systems by using files.

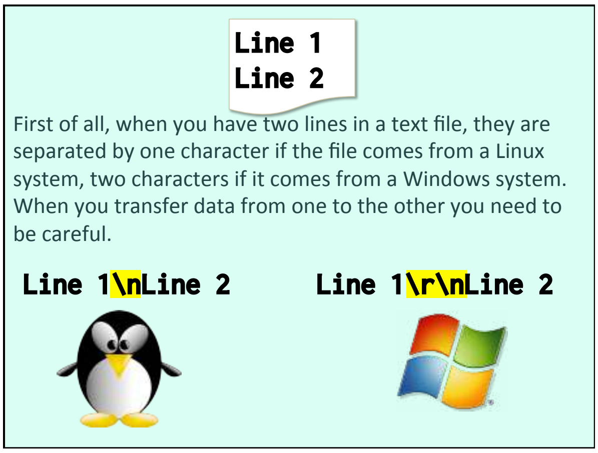
Flickr: Elbfoto

insert into ***table_name***
 ($column_1$, $column_2$, ..., $column_n$)
select col_1, col_2, ..., col_n
from ...

Another way to massively insert data in a table is by inserting the result of a query. Of course, it assumes that the data is already in the database! What usually happens is that data is loaded "as is" in work tables (staging areas) that are badly normalized (they are hardly more than the table image of a file) then dispatched through INSERT ... SELECT ... statements to the well designed tables.



Loading data from a file is typically the type of unglamorous activity that you have to perform quite a lot. It's not always as easy as it should be.

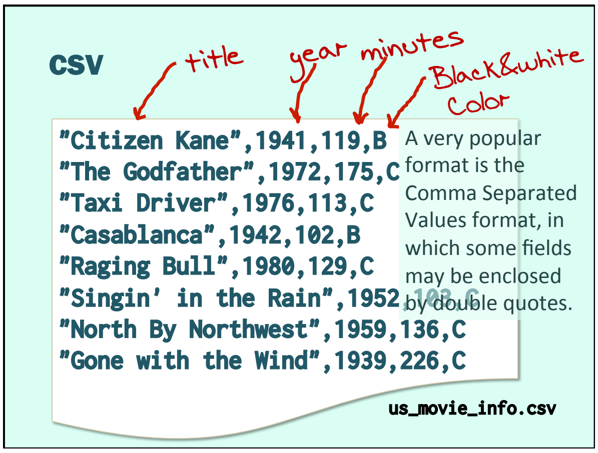


```
Line 1
Line 2
```

First of all, when you have two lines in a text file, they are separated by one character if the file comes from a Linux system, two characters if it comes from a Windows system. When you transfer data from one to the other you need to be careful.

`Line 1\nLine 2`

`Line 1\r\nLine 2`



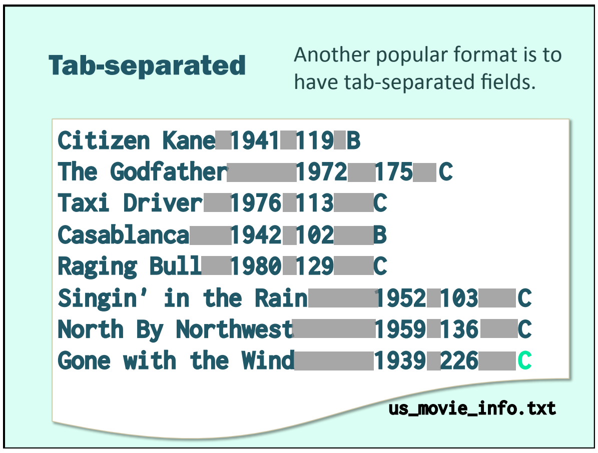
CSV

title year minutes Black&white Color

```
"Citizen Kane",1941,119,B
"The Godfather",1972,175,C
"Taxi Driver",1976,113,C
"Casablanca",1942,102,B
"Raging Bull",1980,129,C
"Singin' in the Rain",1952,103,C
"North By Northwest",1959,136,C
"Gone with the Wind",1939,226,C
```

us_movie_info.csv

A very popular format is the Comma Separated Values format, in which some fields may be enclosed by double quotes.

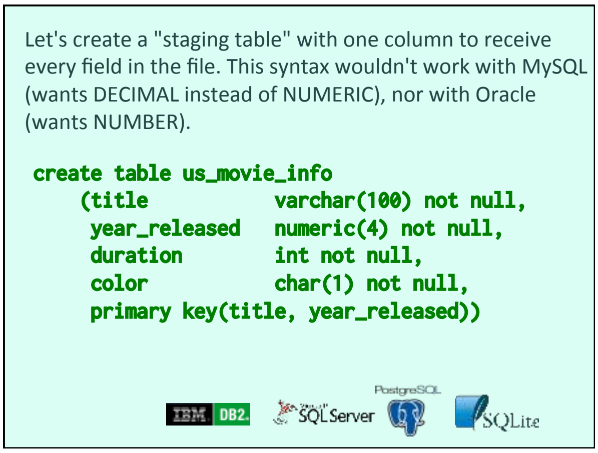


Tab-separated

Another popular format is to have tab-separated fields.

```
Citizen Kane	1941	119	B
The Godfather		1972	175	C
Taxi Driver	1976	113		C
Casablanca		1942	102		B
Raging Bull	1980	129		C
Singin' in the Rain		1952	103		C
North By Northwest		1959	136		C
Gone with the Wind		1939	226		C
```

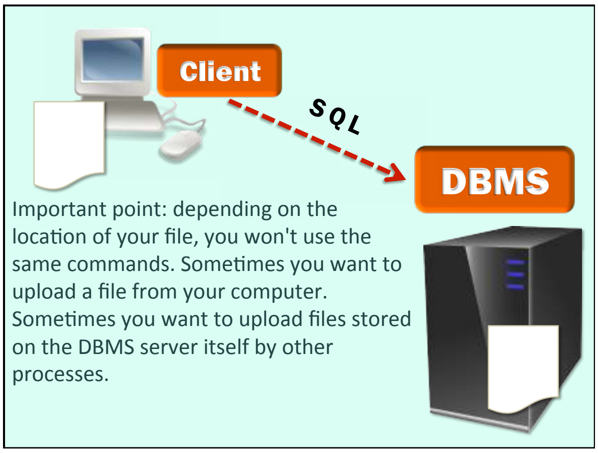
us_movie_info.txt



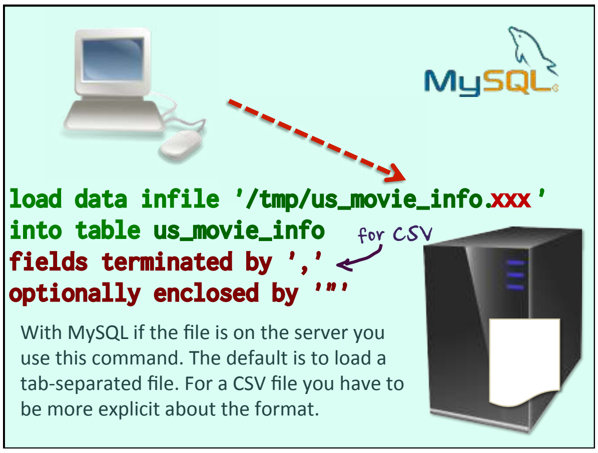
Let's create a "staging table" with one column to receive every field in the file. This syntax wouldn't work with MySQL (wants DECIMAL instead of NUMERIC), nor with Oracle (wants NUMBER).

```
create table us_movie_info
    (title          varchar(100) not null,
     year_released  numeric(4) not null,
     duration       int not null,
     color          char(1) not null,
     primary key(title, year_released))
```

IBM DB2
SQL Server
PostgreSQL
SQLite



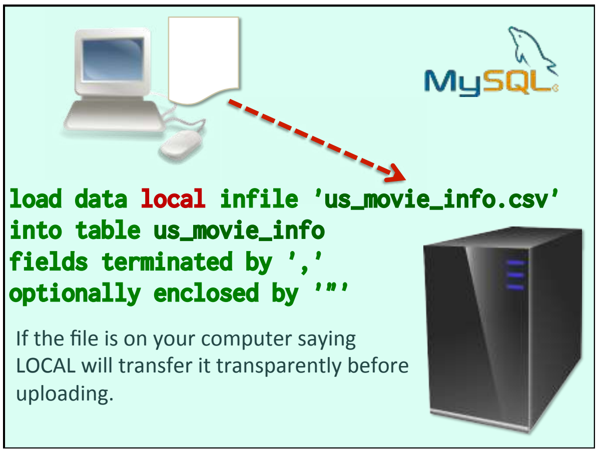
Important point: depending on the location of your file, you won't use the same commands. Sometimes you want to upload a file from your computer. Sometimes you want to upload files stored on the DBMS server itself by other processes.



```
load data infile '/tmp/us_movie_info.xxx '
into table us_movie_info
fields terminated by ','
optionally enclosed by '"'
```

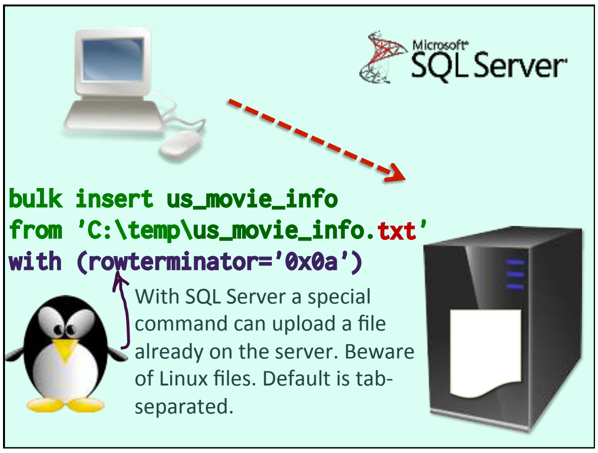
for CSV

With MySQL if the file is on the server you use this command. The default is to load a tab-separated file. For a CSV file you have to be more explicit about the format.



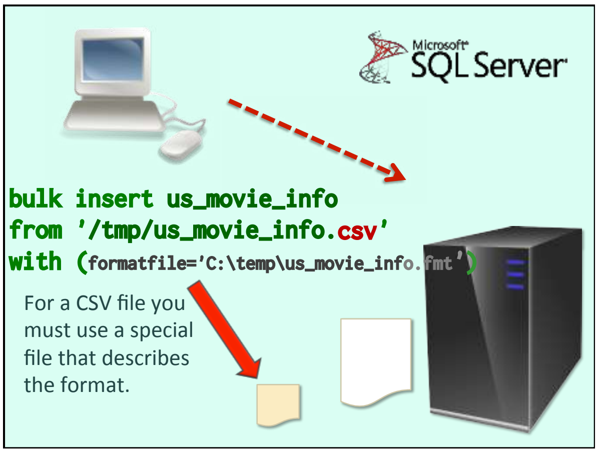
```
load data local infile 'us_movie_info.csv'
into table us_movie_info
fields terminated by ','
optionally enclosed by '"'
```

If the file is on your computer saying LOCAL will transfer it transparently before uploading.



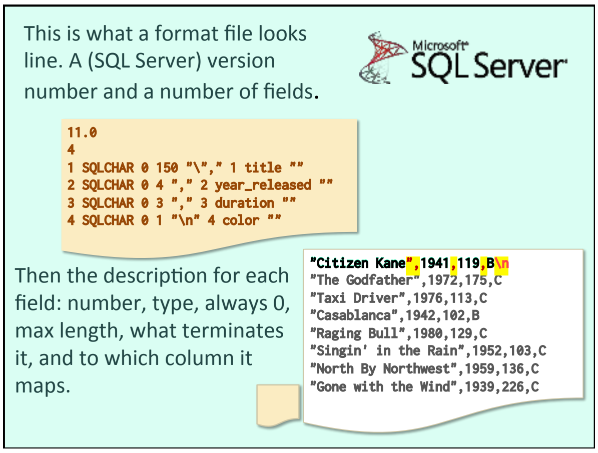
```
bulk insert us_movie_info
from 'C:\temp\us_movie_info.txt'
with (rowterminator='0x0a')
```

With SQL Server a special command can upload a file already on the server. Beware of Linux files. Default is tab-separated.



```
bulk insert us_movie_info
from '/tmp/us_movie_info.csv'
with (formatfile='C:\temp\us_movie_info.fmt')
```

For a CSV file you must use a special file that describes the format.

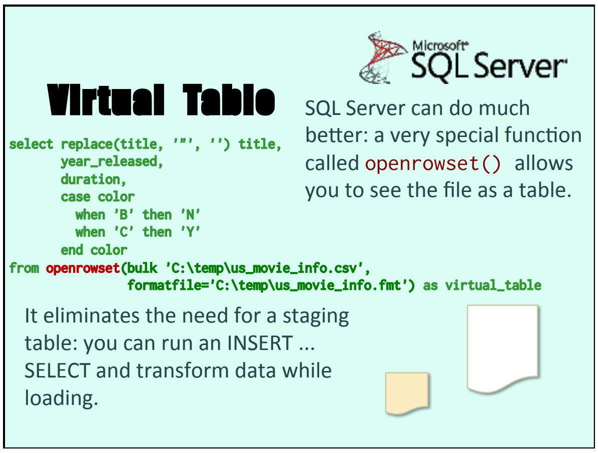


This is what a format file looks line. A (SQL Server) version number and a number of fields.

```
11.0
4
1 SQLCHAR 0 150 "\"," 1 title ""
2 SQLCHAR 0 4 "," 2 year_released ""
3 SQLCHAR 0 3 "," 3 duration ""
4 SQLCHAR 0 1 "\n" 4 color ""
```

Then the description for each field: number, type, always 0, max length, what terminates it, and to which column it maps.

```
"Citizen Kane",1941,119,B\n
"The Godfather",1972,175,C
"Taxi Driver",1976,113,C
"Casablanca",1942,102,B
"Raging Bull",1980,129,C
"Singin' in the Rain",1952,103,C
"North By Northwest",1959,136,C
"Gone with the Wind",1939,226,C
```

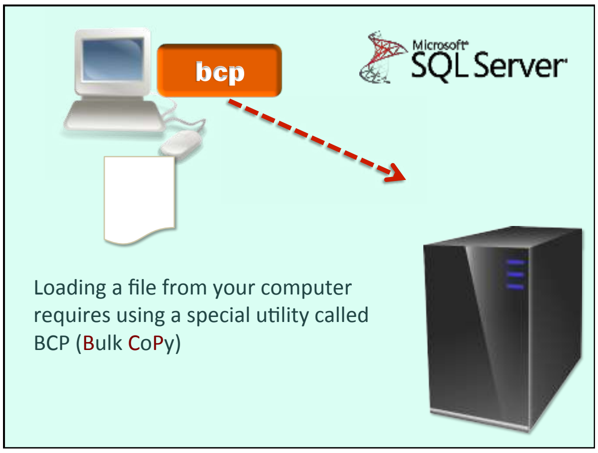


Virtual Table

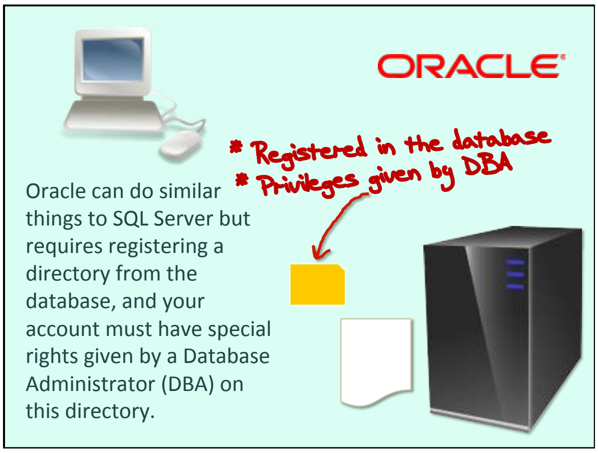
```
select replace(title, '"', '') title,
       year_released,
       duration,
       case color
         when 'B' then 'N'
         when 'C' then 'Y'
       end color
from openrowset(bulk 'C:\temp\us_movie_info.csv',
                formatfile='C:\temp\us_movie_info.fmt') as virtual_table
```

SQL Server can do much better: a very special function called openrowset() allows you to see the file as a table.

It eliminates the need for a staging table: you can run an INSERT ... SELECT and transform data while loading.

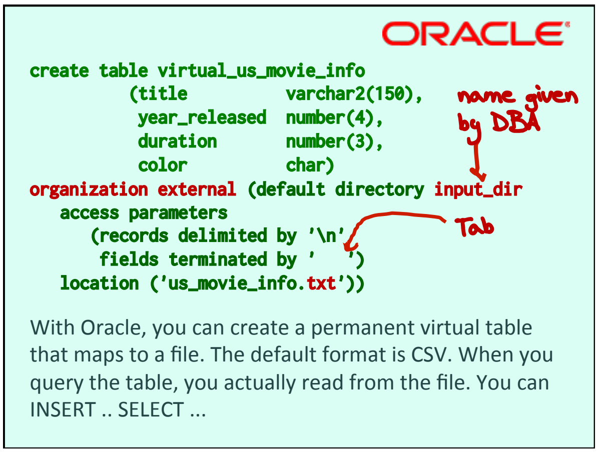


Loading a file from your computer requires using a special utility called BCP (Bulk CoPy)



Oracle can do similar things to SQL Server but requires registering a directory from the database, and your account must have special rights given by a Database Administrator (DBA) on this directory.

* Registered in the database
* Privileges given by DBA

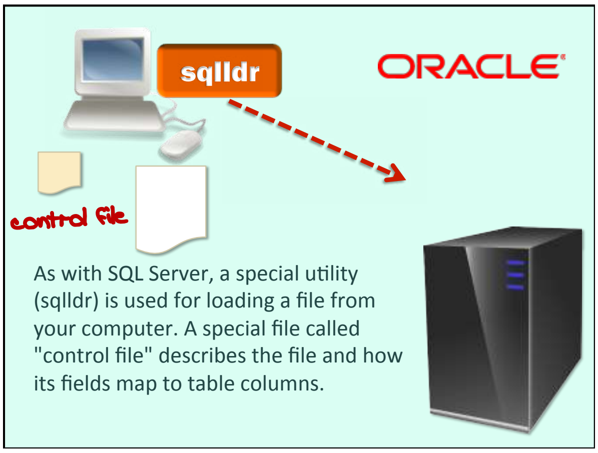


```
create table virtual_us_movie_info
          (title          varchar2(150),
           year_released  number(4),
           duration       number(3),
           color          char)
organization external (default directory input_dir
    access parameters
        (records delimited by '\n'
         fields terminated by '    ')
    location ('us_movie_info.txt'))
```

name given by DBA

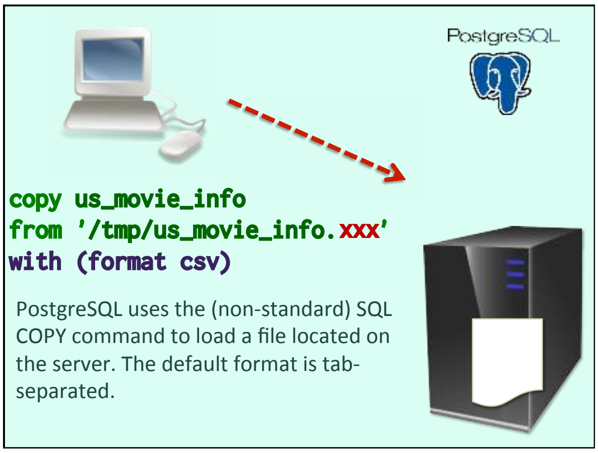
Tab

With Oracle, you can create a permanent virtual table that maps to a file. The default format is CSV. When you query the table, you actually read from the file. You can INSERT .. SELECT ...



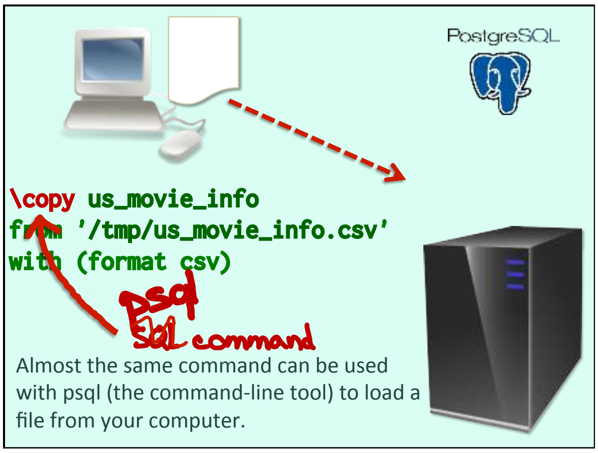
control file

As with SQL Server, a special utility (sqlldr) is used for loading a file from your computer. A special file called "control file" describes the file and how its fields map to table columns.



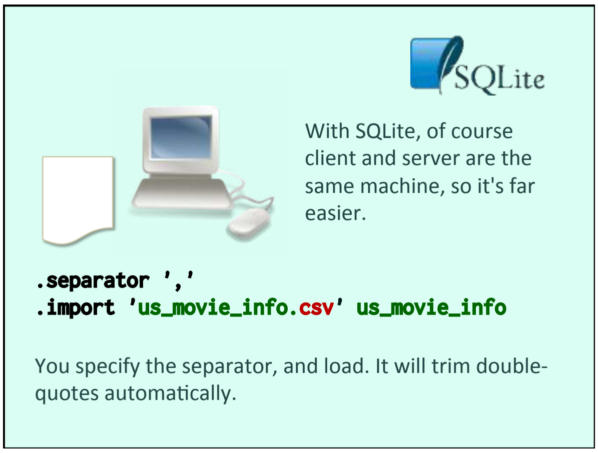
```
copy us_movie_info
from '/tmp/us_movie_info.xxx'
with (format csv)
```

PostgreSQL uses the (non-standard) SQL COPY command to load a file located on the server. The default format is tab-separated.



```
\copy us_movie_info
from '/tmp/us_movie_info.csv'
with (format csv)
```

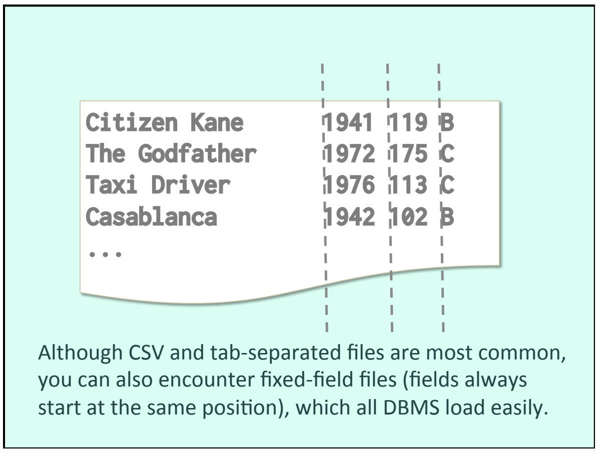
Almost the same command can be used with psql (the command-line tool) to load a file from your computer.



With SQLite, of course client and server are the same machine, so it's far easier.

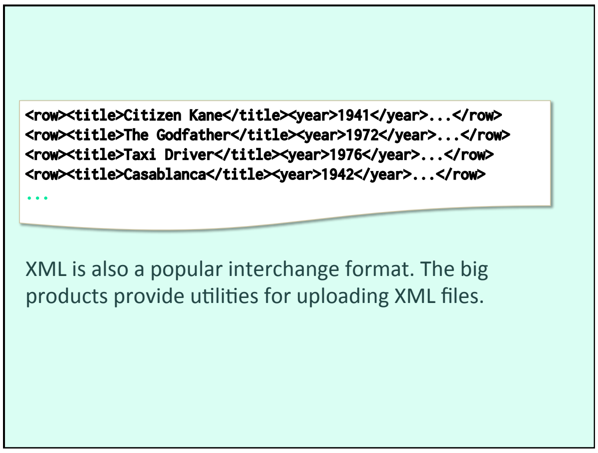
```
.separator ','
.import 'us_movie_info.csv' us_movie_info
```

You specify the separator, and load. It will trim double-quotes automatically.



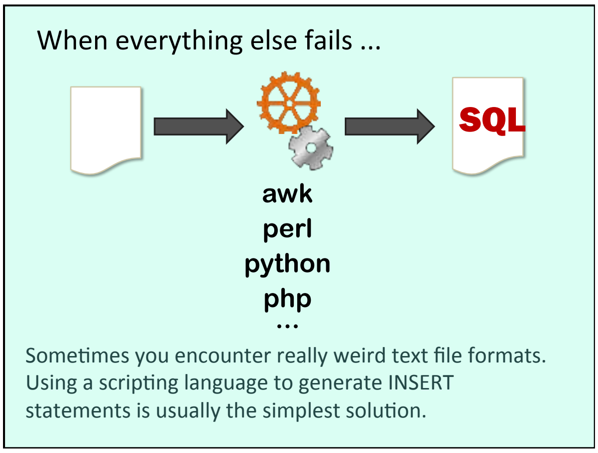
```
Citizen Kane      1941 119 B
The Godfather     1972 175 C
Taxi Driver       1976 113 C
Casablanca        1942 102 B
...
```

Although CSV and tab-separated files are most common, you can also encounter fixed-field files (fields always start at the same position), which all DBMS load easily.



```
<row><title>Citizen Kane</title><year>1941</year>...</row>
<row><title>The Godfather</title><year>1972</year>...</row>
<row><title>Taxi Driver</title><year>1976</year>...</row>
<row><title>Casablanca</title><year>1942</year>...</row>
...
```

XML is also a popular interchange format. The big products provide utilities for uploading XML files.

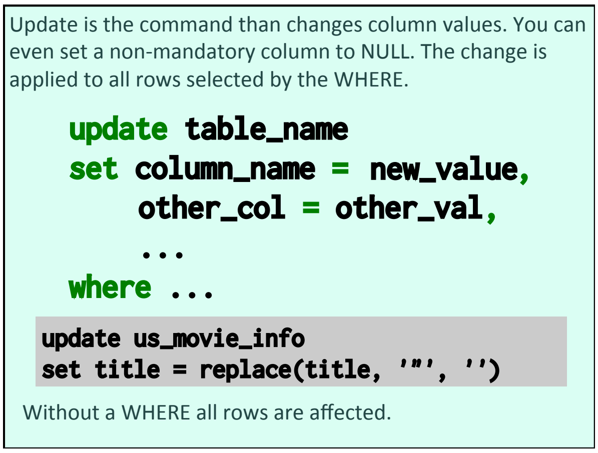


When everything else fails ...

Sometimes you encounter really weird text file formats. Using a scripting language to generate INSERT statements is usually the simplest solution.

Things change ...

We have talked about inserting data, lets' now see how we can update what is in the database.

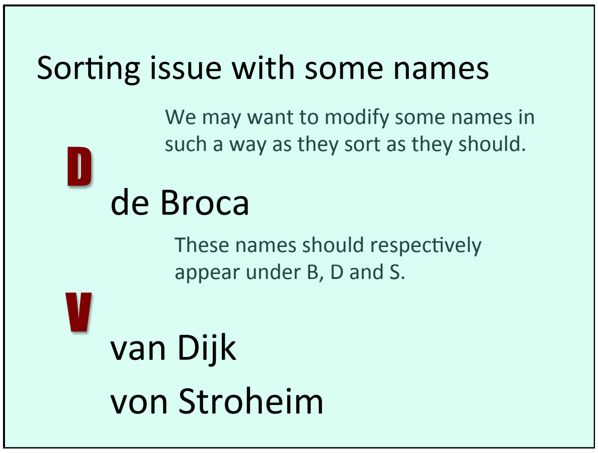


Update is the command than changes column values. You can even set a non-mandatory column to NULL. The change is applied to all rows selected by the WHERE.

```
update table_name
set column_name = new_value,
    other_col = other_val,
    ...
where ...
```

```
update us_movie_info
set title = replace(title, '"', '')
```

Without a WHERE all rows are affected.



Sorting issue with some names

We may want to modify some names in such a way as they sort as they should.

D

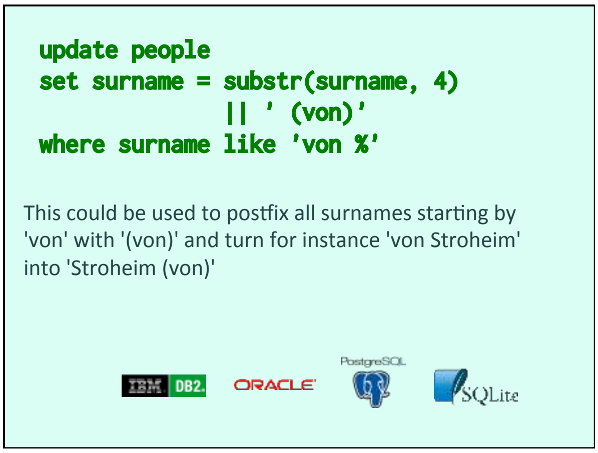
de Broca

These names should respectively appear under B, D and S.

V

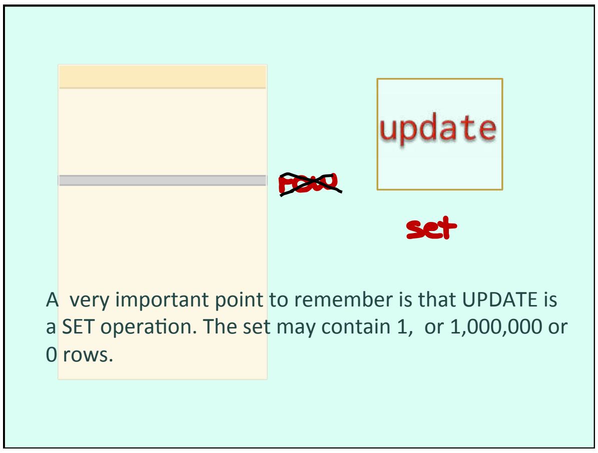
van Dijk

von Stroheim

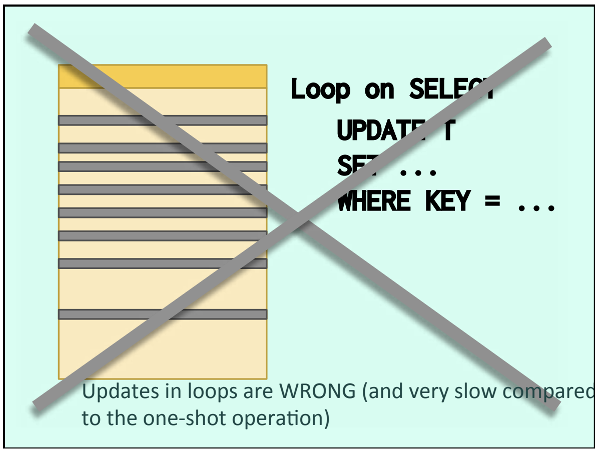


```
update people
set surname = substr(surname, 4)
              || ' (von)'
where surname like 'von %'
```

This could be used to postfix all surnames starting by 'von' with '(von)' and turn for instance 'von Stroheim' into 'Stroheim (von)'

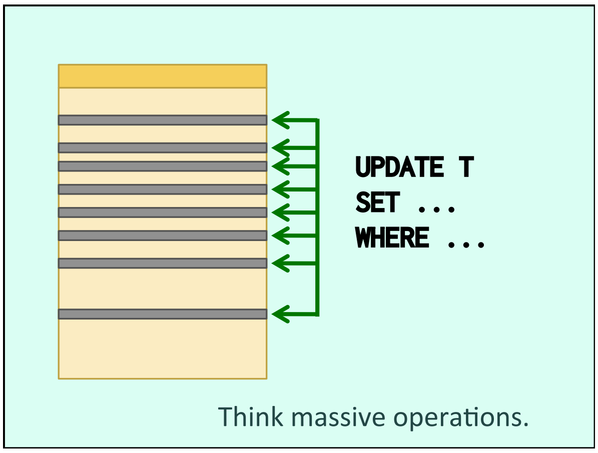


A very important point to remember is that UPDATE is a SET operation. The set may contain 1, or 1,000,000 or 0 rows.



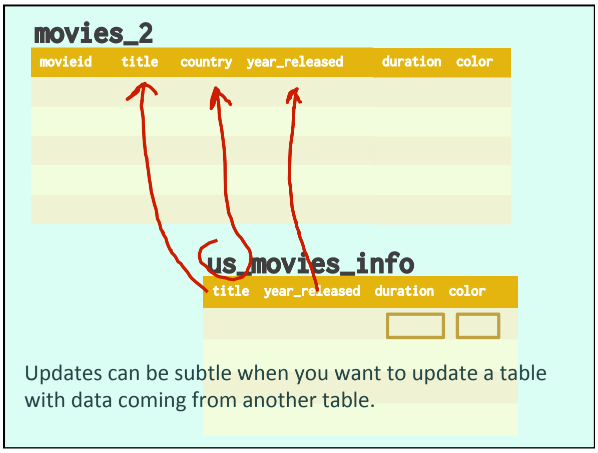
```
~~Loop on SELECT
    UPDATE T
    SET ...
    WHERE KEY = ...~~
```

Updates in loops are WRONG (and very slow compared to the one-shot operation)



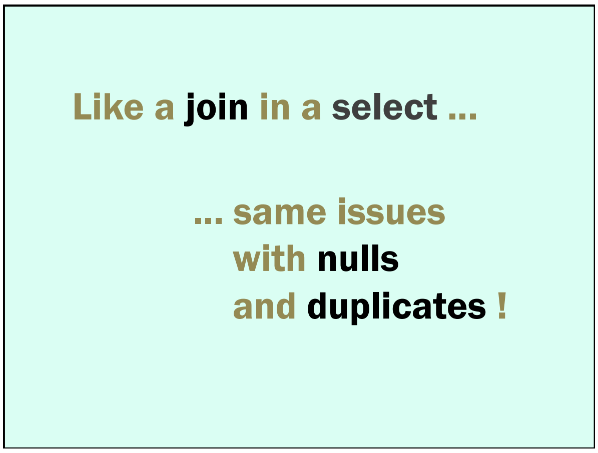
UPDATE T
SET ...
WHERE ...

Think massive operations.



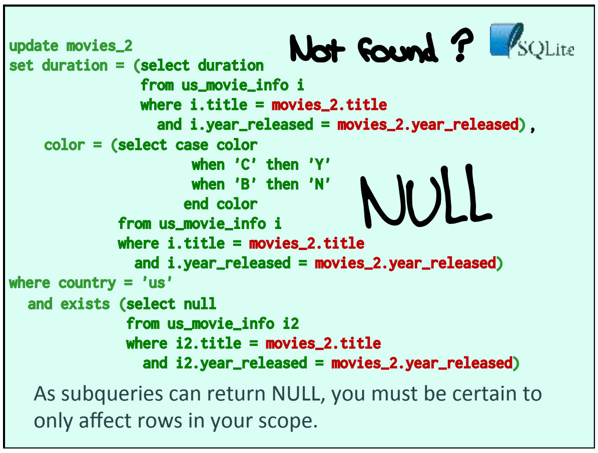
movies_2
movieid title country year_released duration color
us_movies_info
title year_released duration color

Updates can be subtle when you want to update a table with data coming from another table.



Like a **join** in a **select** ...

... same issues
with **nulls**
and **duplicates** !

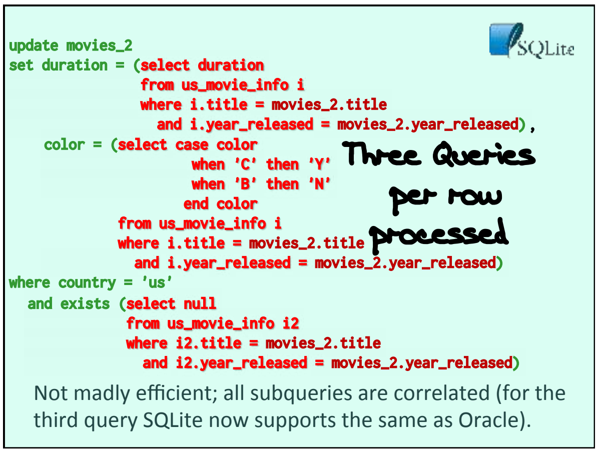


Not found ?

```
update movies_2
set duration = (select duration
                from us_movie_info i
                where i.title = movies_2.title
                  and i.year_released = movies_2.year_released),
    color = (select case color
                      when 'C' then 'Y'
                      when 'B' then 'N'
                     end color
             from us_movie_info i
             where i.title = movies_2.title
               and i.year_released = movies_2.year_released)
where country = 'us'
  and exists (select null
              from us_movie_info i2
              where i2.title = movies_2.title
                and i2.year_released = movies_2.year_released)
```

NULL

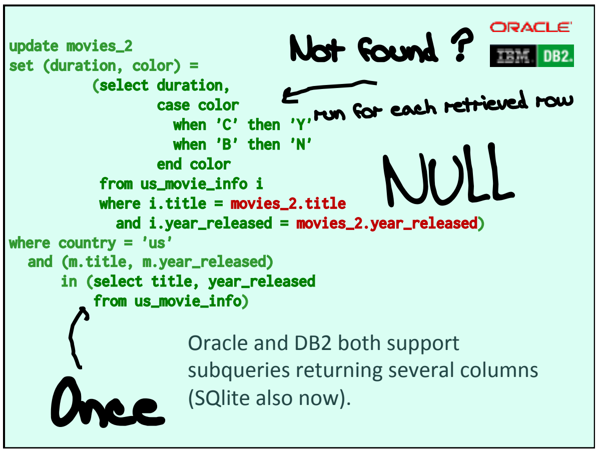
As subqueries can return NULL, you must be certain to only affect rows in your scope.



```
update movies_2
set duration = (select duration
                from us_movie_info i
                where i.title = movies_2.title
                  and i.year_released = movies_2.year_released),
    color = (select case color
                      when 'C' then 'Y'
                      when 'B' then 'N'
                    end color
             from us_movie_info i
             where i.title = movies_2.title
               and i.year_released = movies_2.year_released)
where country = 'us'
  and exists (select null
              from us_movie_info i2
              where i2.title = movies_2.title
                and i2.year_released = movies_2.year_released)
```

Three Queries per row processed

Not madly efficient; all subqueries are correlated (for the third query SQLite now supports the same as Oracle).



```
update movies_2
set (duration, color) =
         (select duration,
                 case color
                   when 'C' then 'Y'
                   when 'B' then 'N'
                 end color
          from us_movie_info i
          where i.title = movies_2.title
            and i.year_released = movies_2.year_released)
where country = 'us'
  and (m.title, m.year_released)
      in (select title, year_released
          from us_movie_info)
```

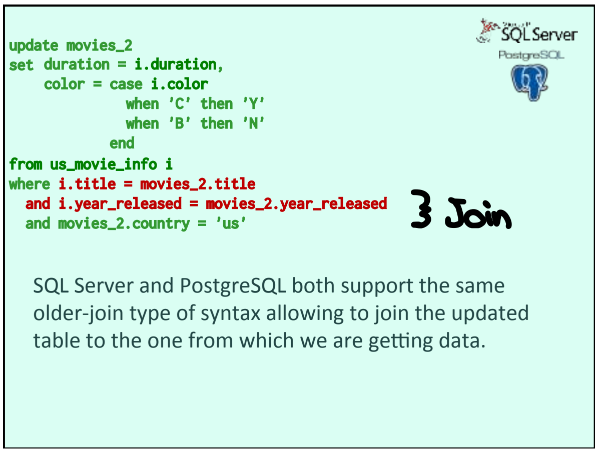
Not found ?

run for each retrieved row

NULL

Once

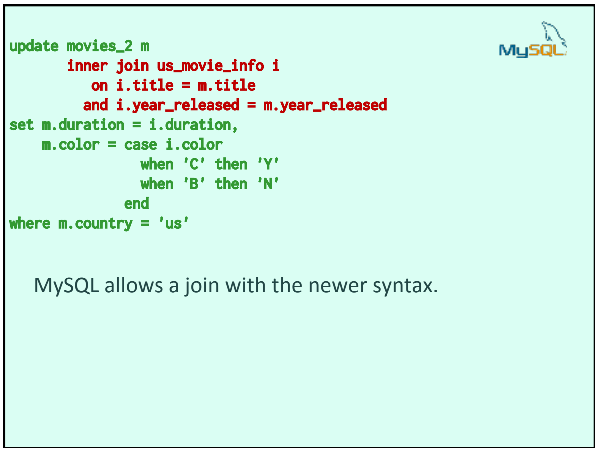
Oracle and DB2 both support subqueries returning several columns (SQlite also now).



```
update movies_2
set duration = i.duration,
    color = case i.color
              when 'C' then 'Y'
              when 'B' then 'N'
            end
from us_movie_info i
where i.title = movies_2.title
  and i.year_released = movies_2.year_released
  and movies_2.country = 'us'
```

3 Join

SQL Server and PostgreSQL both support the same older-join type of syntax allowing to join the updated table to the one from which we are getting data.



```
update movies_2 m
       inner join us_movie_info i
         on i.title = m.title
         and i.year_released = m.year_released
set m.duration = i.duration,
    m.color = case i.color
                when 'C' then 'Y'
                when 'B' then 'N'
              end
where m.country = 'us'
```

MySQL allows a join with the newer syntax.

What can happen when join conditions are

WRONG?

When you have a SELECT wrong, it only affects your query. When you have an UPDATE wrong, you can corrupt the database and later correct queries on wrong data will return wrong results.
So you really need to be extra careful.

movies_2 **us_movie_info**

title ⟷ **title**

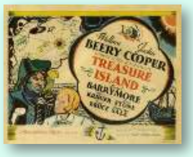
year_released ⟷ year_released

country = 'us'

Imagine for instance that we forget the join on the year and that we have remakes. What will happen?

movies_2

us_movie_info



Let's first say that we only have remakes in the table that we update.

Running a SELECT shows what happens.

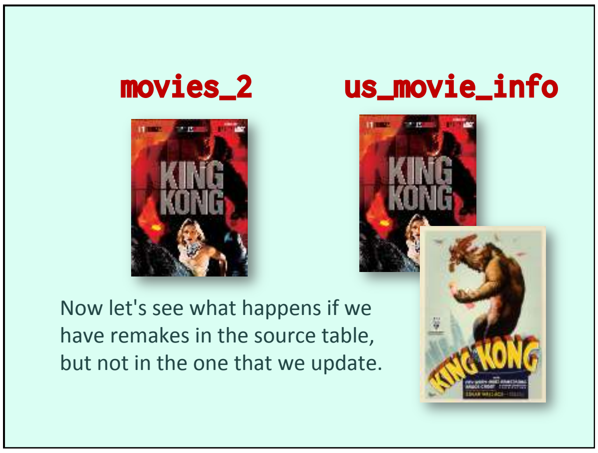
```
select m.title, m.year_released,
       i.year_released, i.duration, i.color
from movies_2 m
     inner join us_movie_info i
        on i.title = m.title
where m.title like 'Treasure%'
```

One row from the source table will be associated with both films.

```
title              year_released    year_released    duration   color
------------------ ---------------- ---------------- ---------- ------
Treasure Island                1934             1934        103 B
Treasure Island                1950             1934        103 B
```

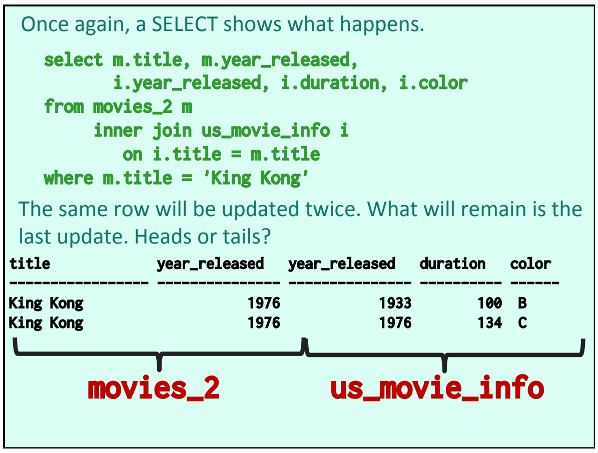
movies_2 **us_movie_info**

We'll set wrong information for the 1950 film.



movies_2 **us_movie_info**

Now let's see what happens if we have remakes in the source table, but not in the one that we update.

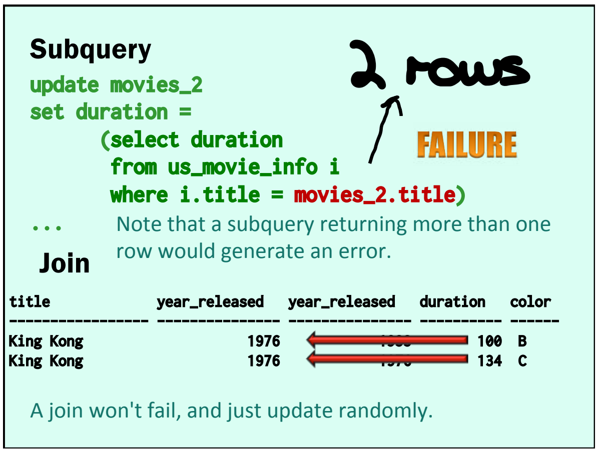


Once again, a SELECT shows what happens.

```
select m.title, m.year_released,
       i.year_released, i.duration, i.color
from movies_2 m
     inner join us_movie_info i
        on i.title = m.title
where m.title = 'King Kong'
```

The same row will be updated twice. What will remain is the last update. Heads or tails?

```
title              year_released   year_released   duration   color
------------------ --------------- --------------- ---------- ------
King Kong                     1976            1933        100  B
King Kong                     1976            1976        134  C
```



Subquery

```
update movies_2
set duration =
      (select duration
       from us_movie_info i
       where i.title = movies_2.title)
...
```

2 rows

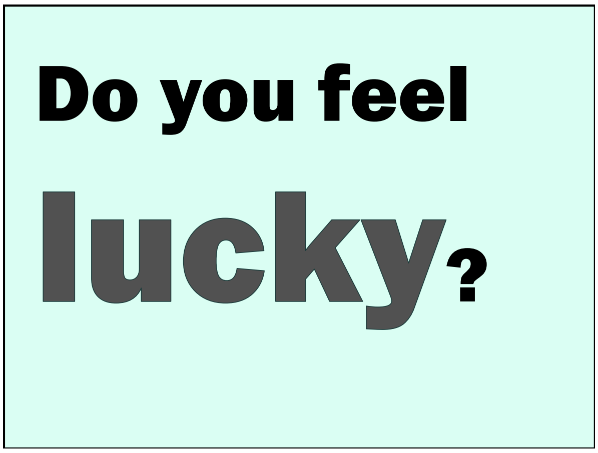
FAILURE

Note that a subquery returning more than one row would generate an error.

Join

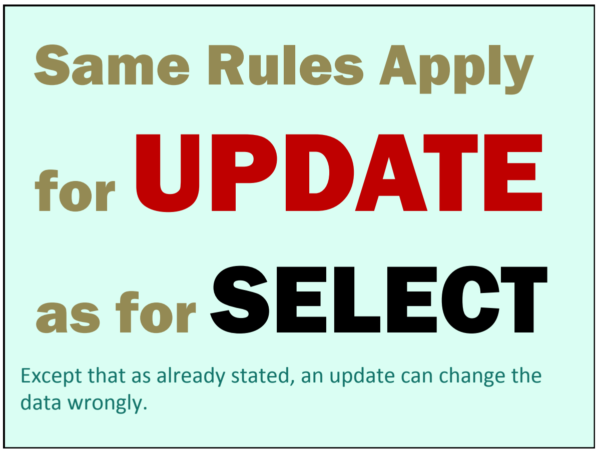
```
title              year_released   year_released   duration   color
------------------ --------------- --------------- ---------- ------
King Kong                     1976     [illegible]        100  B
King Kong                     1976     [illegible]        134  C
```

A join won't fail, and just update randomly.



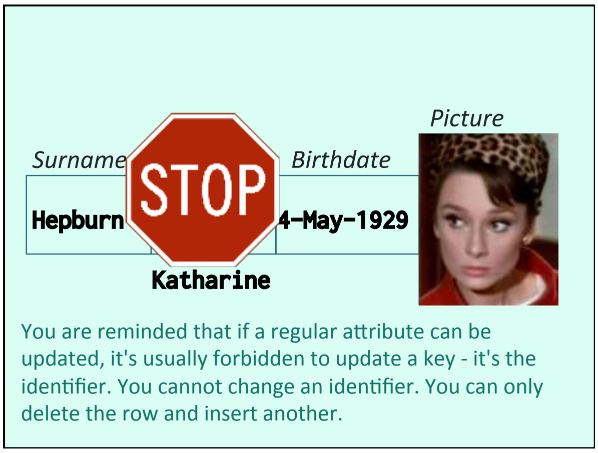
Do you feel

lucky?

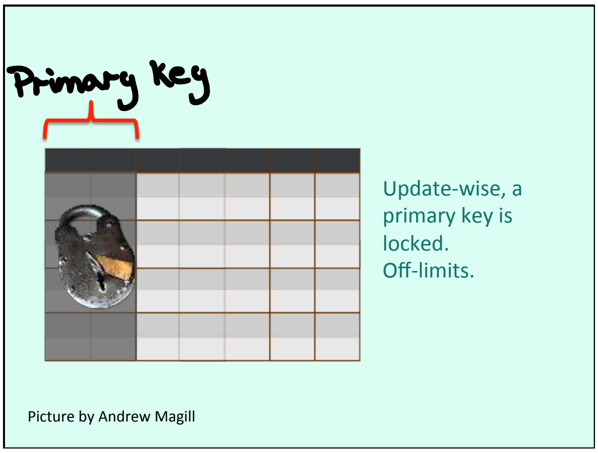


Same Rules Apply for UPDATE as for SELECT

Except that as already stated, an update can change the data wrongly.

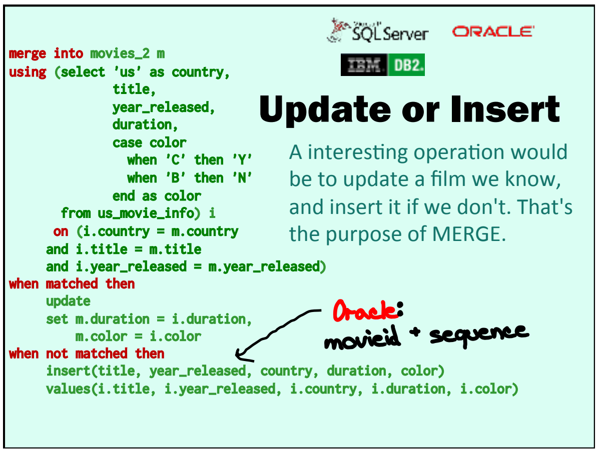


You are reminded that if a regular attribute can be updated, it's usually forbidden to update a key - it's the identifier. You cannot change an identifier. You can only delete the row and insert another.



Update-wise, a primary key is locked. Off-limits.

Picture by Andrew Magill



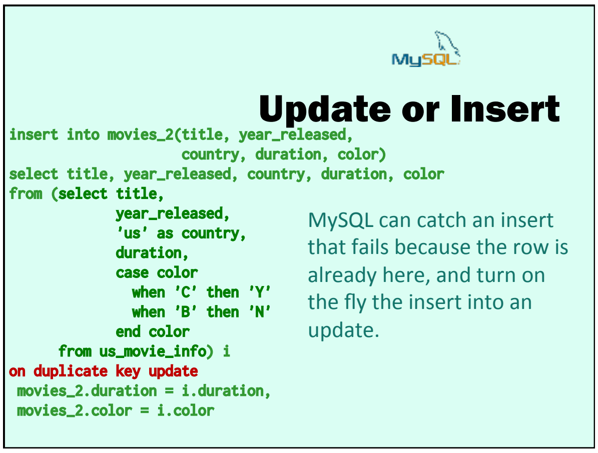
SQL Server
ORACLE
IBM DB2.

Update or Insert

A interesting operation would be to update a film we know, and insert it if we don't. That's the purpose of MERGE.

```
merge into movies_2 m
using (select 'us' as country,
              title,
              year_released,
              duration,
              case color
                when 'C' then 'Y'
                when 'B' then 'N'
              end as color
       from us_movie_info) i
      on (i.country = m.country
     and i.title = m.title
     and i.year_released = m.year_released)
when matched then
     update
     set m.duration = i.duration,
         m.color = i.color
when not matched then
     insert(title, year_released, country, duration, color)
     values(i.title, i.year_released, i.country, i.duration, i.color)
```

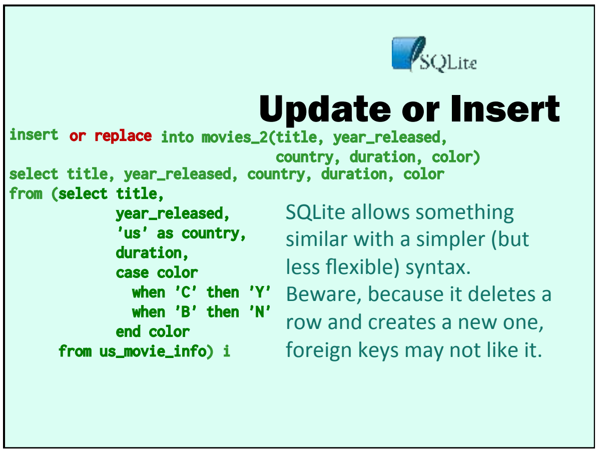
Oracle:
movieid + sequence



Update or Insert

```
insert into movies_2(title, year_released,
                     country, duration, color)
select title, year_released, country, duration, color
from (select title,
             year_released,
             'us' as country,
             duration,
             case color
               when 'C' then 'Y'
               when 'B' then 'N'
             end color
      from us_movie_info) i
on duplicate key update
 movies_2.duration = i.duration,
 movies_2.color = i.color
```

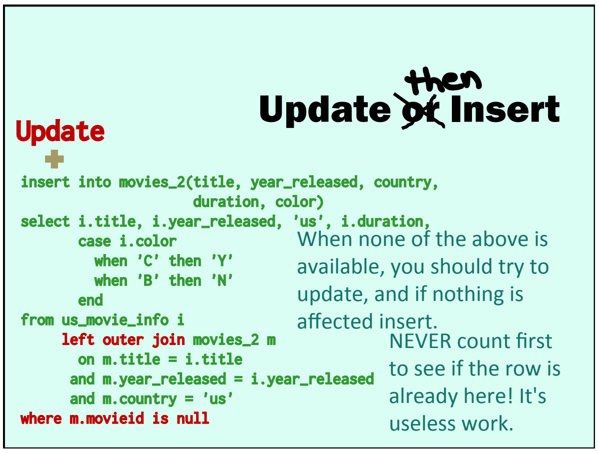
MySQL can catch an insert that fails because the row is already here, and turn on the fly the insert into an update.



Update or Insert

```
insert  or replace into movies_2(title, year_released,
                                 country, duration, color)
select title, year_released, country, duration, color
from (select title,
             year_released,
             'us' as country,
             duration,
             case color
               when 'C' then 'Y'
               when 'B' then 'N'
             end color
      from us_movie_info) i
```

SQLite allows something similar with a simpler (but less flexible) syntax. Beware, because it deletes a row and creates a new one, foreign keys may not like it.



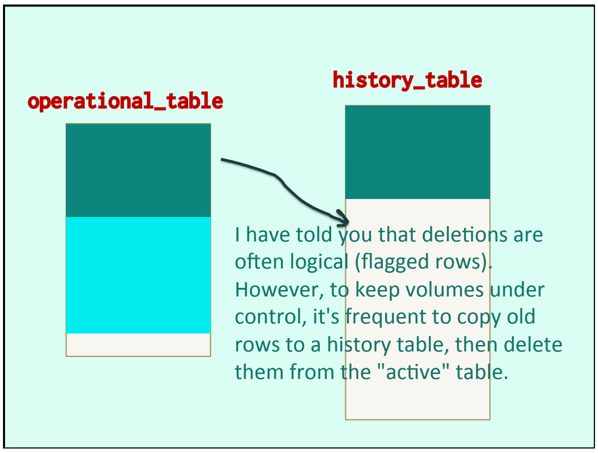
Update ~~or~~ then Insert

Update
+

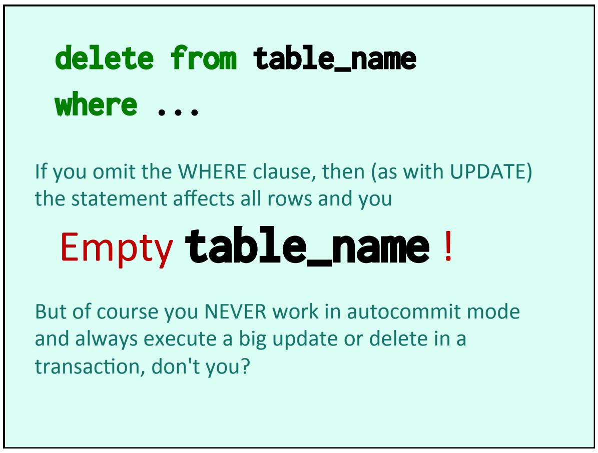
```
insert into movies_2(title, year_released, country,
                     duration, color)
select i.title, i.year_released, 'us', i.duration,
       case i.color
         when 'C' then 'Y'
         when 'B' then 'N'
       end
from us_movie_info i
     left outer join movies_2 m
       on m.title = i.title
      and m.year_released = i.year_released
      and m.country = 'us'
where m.movieid is null
```

When none of the above is available, you should try to update, and if nothing is affected insert.

NEVER count first to see if the row is already here! It's useless work.



I have told you that deletions are often logical (flagged rows). However, to keep volumes under control, it's frequent to copy old rows to a history table, then delete them from the "active" table.

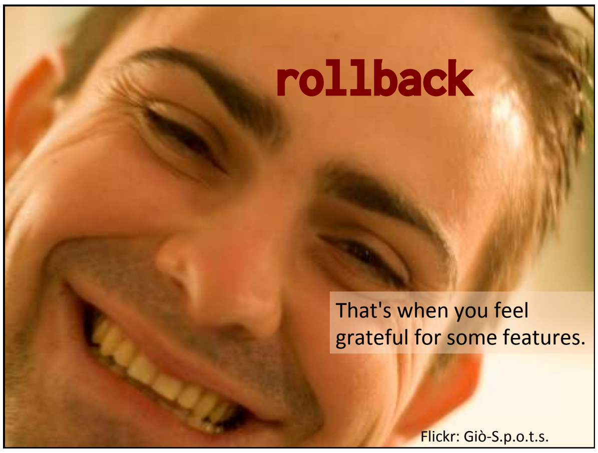


```
delete from table_name
where ...
```

If you omit the WHERE clause, then (as with UPDATE) the statement affects all rows and you

Empty **table_name** !

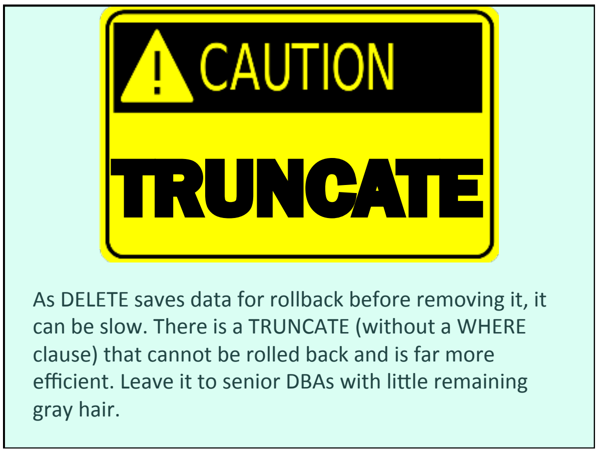
But of course you NEVER work in autocommit mode and always execute a big update or delete in a transaction, don't you?



rollback

That's when you feel grateful for some features.

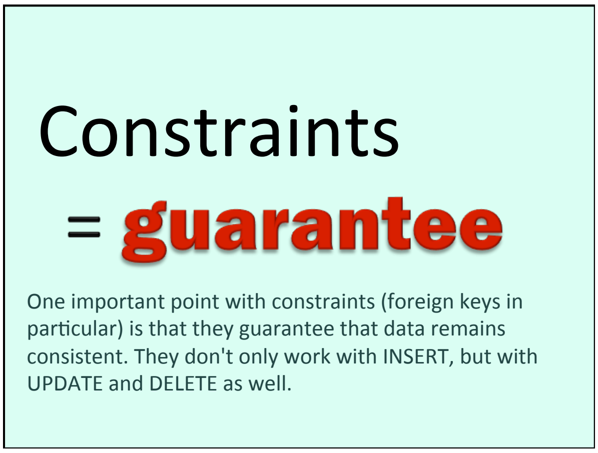
Flickr: Giò-S.p.o.t.s.



CAUTION

TRUNCATE

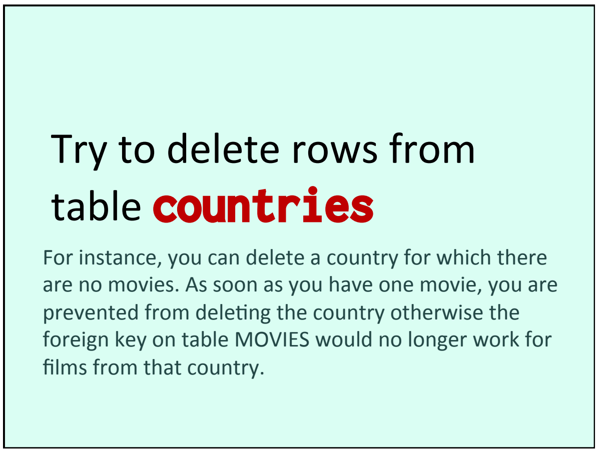
As DELETE saves data for rollback before removing it, it can be slow. There is a TRUNCATE (without a WHERE clause) that cannot be rolled back and is far more efficient. Leave it to senior DBAs with little remaining gray hair.



Constraints

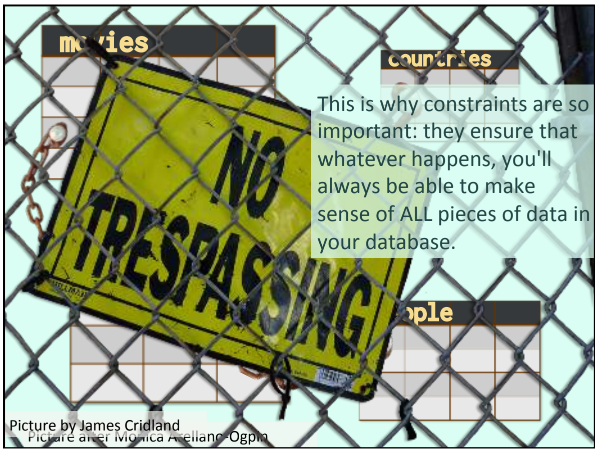
= **guarantee**

One important point with constraints (foreign keys in particular) is that they guarantee that data remains consistent. They don't only work with INSERT, but with UPDATE and DELETE as well.



Try to delete rows from table **countries**

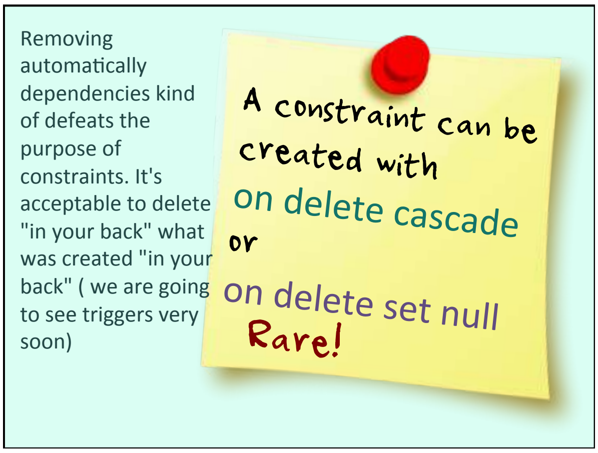
For instance, you can delete a country for which there are no movies. As soon as you have one movie, you are prevented from deleting the country otherwise the foreign key on table MOVIES would no longer work for films from that country.



This is why constraints are so important: they ensure that whatever happens, you'll always be able to make sense of ALL pieces of data in your database.

Picture by James Cridland

Picture after Monica Arellano-Ogpin



Removing automatically dependencies kind of defeats the purpose of constraints. It's acceptable to delete "in your back" what was created "in your back" (we are going to see triggers very soon)

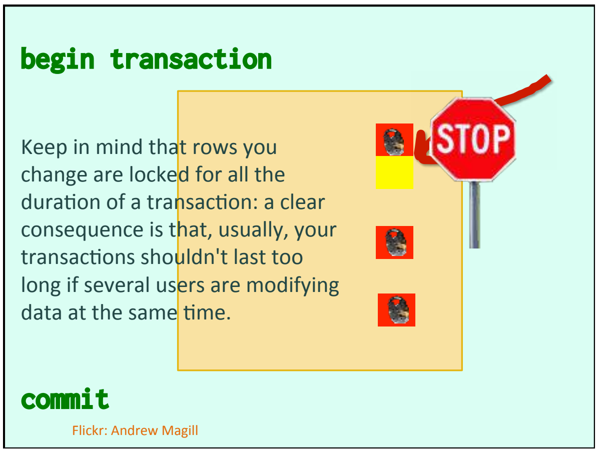
A constraint can be created with

on delete cascade

or

on delete set null

Rare!

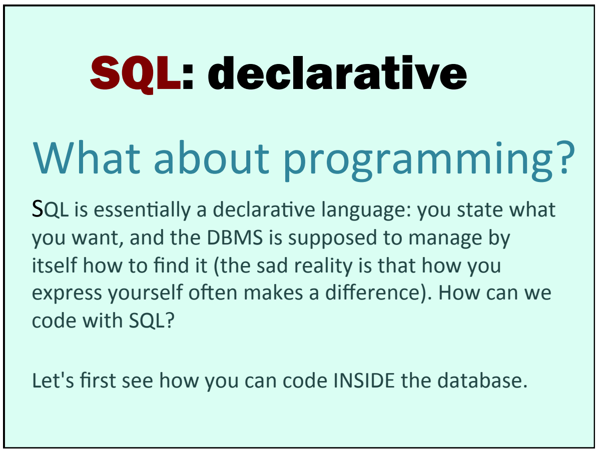


begin transaction

Keep in mind that rows you change are locked for all the duration of a transaction: a clear consequence is that, usually, your transactions shouldn't last too long if several users are modifying data at the same time.

commit

Flickr: Andrew Magill

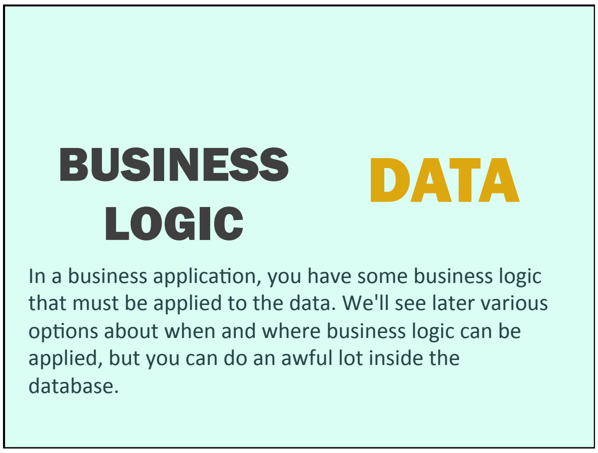


SQL: declarative

What about programming?

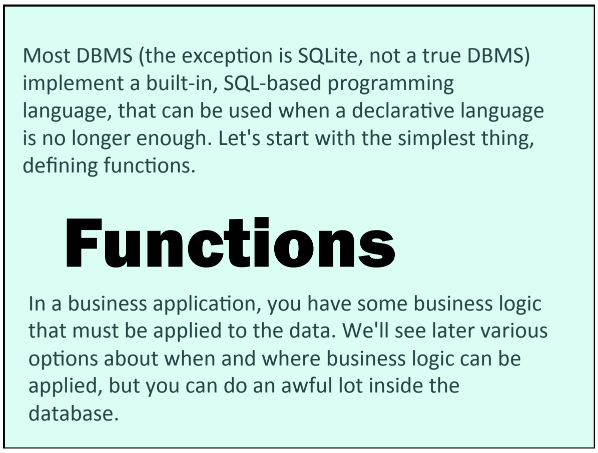
SQL is essentially a declarative language: you state what you want, and the DBMS is supposed to manage by itself how to find it (the sad reality is that how you express yourself often makes a difference). How can we code with SQL?

Let's first see how you can code INSIDE the database.



BUSINESS LOGIC **DATA**

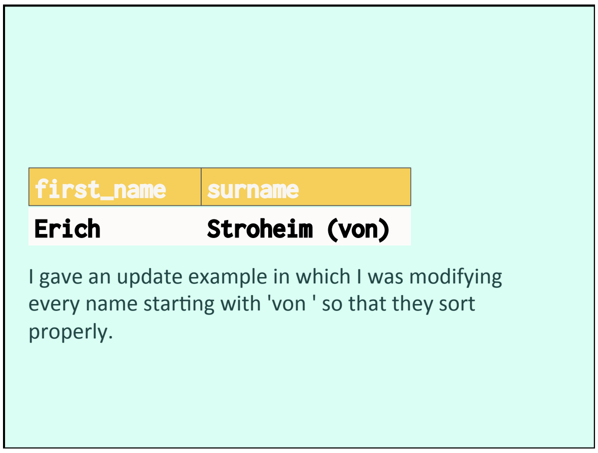
In a business application, you have some business logic that must be applied to the data. We'll see later various options about when and where business logic can be applied, but you can do an awful lot inside the database.



Most DBMS (the exception is SQLite, not a true DBMS) implement a built-in, SQL-based programming language, that can be used when a declarative language is no longer enough. Let's start with the simplest thing, defining functions.

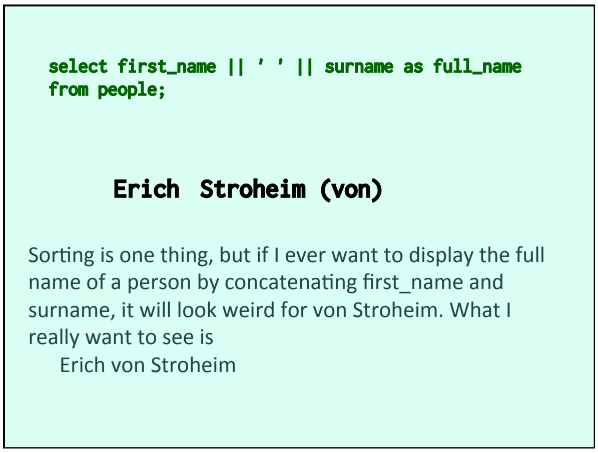
Functions

In a business application, you have some business logic that must be applied to the data. We'll see later various options about when and where business logic can be applied, but you can do an awful lot inside the database.



first_name	surname
Erich	Stroheim (von)

I gave an update example in which I was modifying every name starting with 'von ' so that they sort properly.

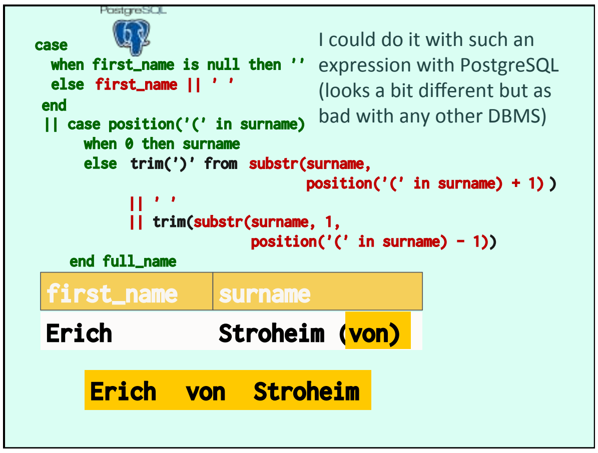


```
select first_name || ' ' || surname as full_name
from people;
```

```
Erich  Stroheim (von)
```

Sorting is one thing, but if I ever want to display the full name of a person by concatenating first_name and surname, it will look weird for von Stroheim. What I really want to see is

Erich von Stroheim



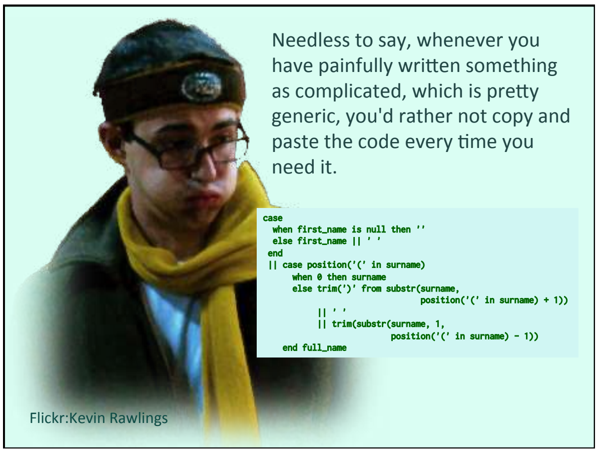
PostgreSQL

```
case
  when first_name is null then ''
  else first_name || ' '
 end
 || case position('(' in surname)
      when 0 then surname
      else trim(')' from substr(surname,
                                  position('(' in surname) + 1) )
           || ' '
           || trim(substr(surname, 1,
                            position('(' in surname) - 1))
    end full_name
```

I could do it with such an expression with PostgreSQL (looks a bit different but as bad with any other DBMS)

first_name	surname
Erich	Stroheim (von)

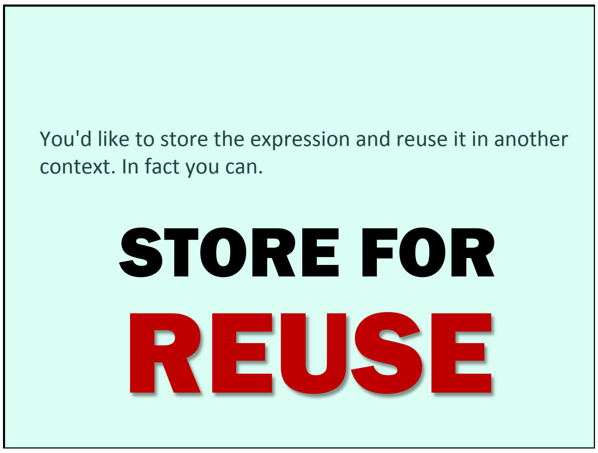
Erich von Stroheim



Needless to say, whenever you have painfully written something as complicated, which is pretty generic, you'd rather not copy and paste the code every time you need it.

```
case
  when first_name is null then ''
  else first_name || ' '
 end
 || case position('(' in surname)
      when 0 then surname
      else trim(')' from substr(surname,
                                  position('(' in surname) + 1))
           || ' '
           || trim(substr(surname, 1,
                            position('(' in surname) - 1))
    end full_name
```

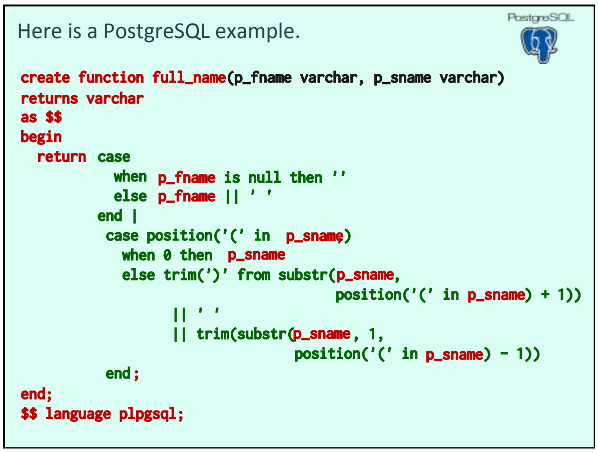
Flickr:Kevin Rawlings



You'd like to store the expression and reuse it in another context. In fact you can.

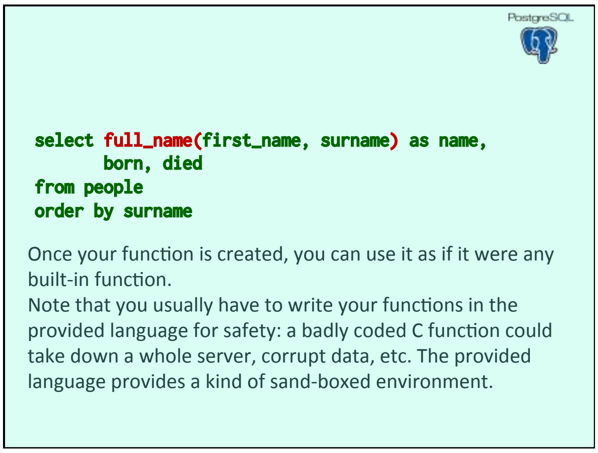
STORE FOR

REUSE



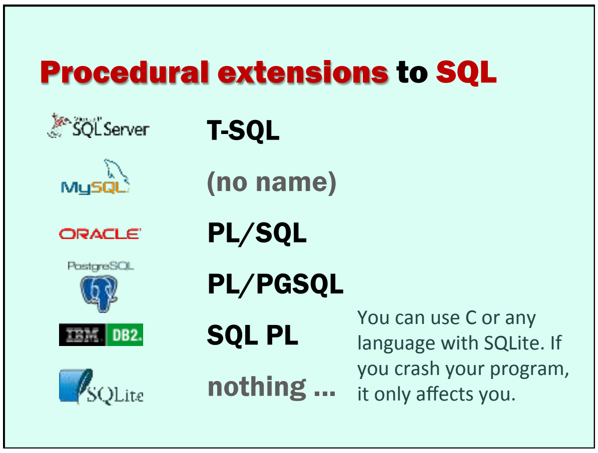
Here is a PostgreSQL example.

```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
  return case
           when p_fname is null then ''
           else p_fname || ' '
         end |
          case position('(' in  p_sname)
            when 0 then  p_sname
            else trim(')' from substr(p_sname,
                                      position('(' in p_sname) + 1))
                  || ' '
                  || trim(substr(p_sname, 1,
                                 position('(' in p_sname) - 1))
          end;
end;
$$ language plpgsql;
```



```
select full_name(first_name, surname) as name,
       born, died
from people
order by surname
```

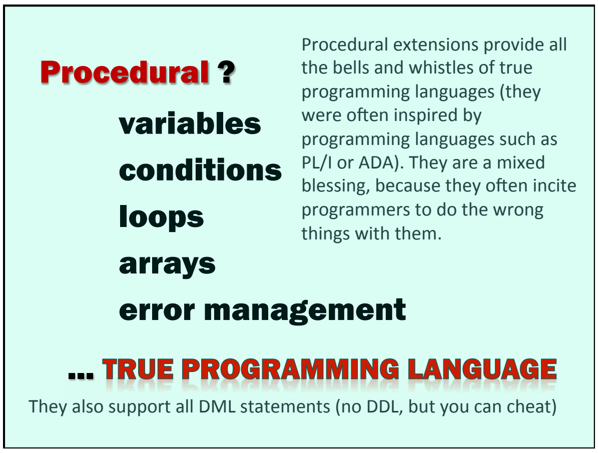
Once your function is created, you can use it as if it were any built-in function.
Note that you usually have to write your functions in the provided language for safety: a badly coded C function could take down a whole server, corrupt data, etc. The provided language provides a kind of sand-boxed environment.



Procedural extensions to SQL

SQL Server	T-SQL
MySQL	(no name)
ORACLE	PL/SQL
PostgreSQL	PL/PGSQL
IBM DB2	SQL PL
SQLite	nothing ...

You can use C or any language with SQLite. If you crash your program, it only affects you.



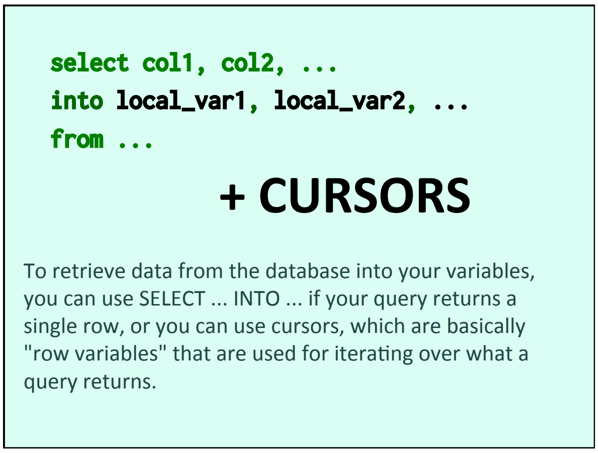
Procedural ?

variables
conditions
loops
arrays
error management

Procedural extensions provide all the bells and whistles of true programming languages (they were often inspired by programming languages such as PL/I or ADA). They are a mixed blessing, because they often incite programmers to do the wrong things with them.

... TRUE PROGRAMMING LANGUAGE

They also support all DML statements (no DDL, but you can cheat)



```
select col1, col2, ...
into local_var1, local_var2, ...
from ...
```

+ CURSORS

To retrieve data from the database into your variables, you can use SELECT ... INTO ... if your query returns a single row, or you can use cursors, which are basically "row variables" that are used for iterating over what a query returns.

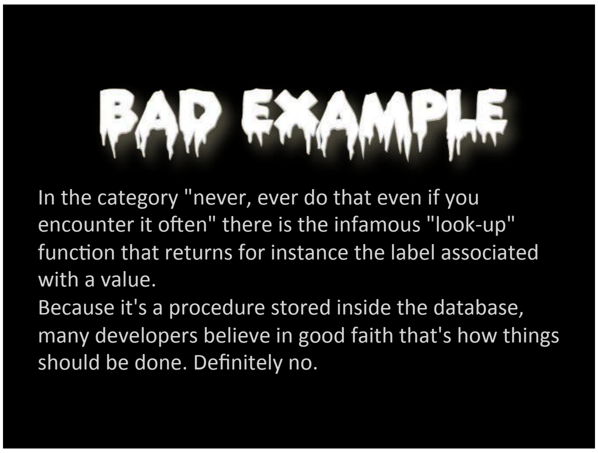
Cultural mismatch

row-by-row

set processing

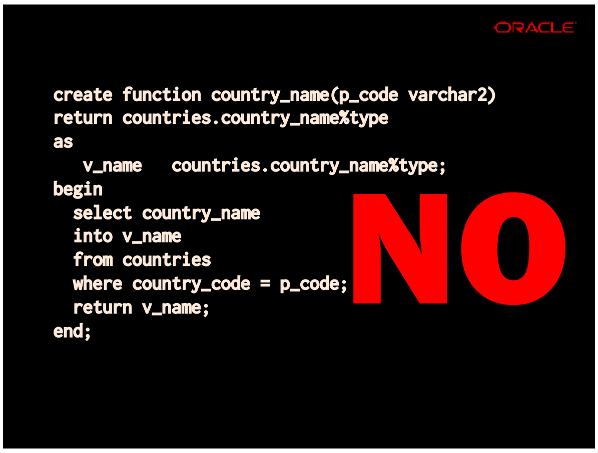
And here we have a problem, because there is a big cultural gap between the relational mindset and procedural processing.

Flickr: Jeff Sandquist



BAD EXAMPLE

In the category "never, ever do that even if you encounter it often" there is the infamous "look-up" function that returns for instance the label associated with a value.
Because it's a procedure stored inside the database, many developers believe in good faith that's how things should be done. Definitely no.



```
create function country_name(p_code varchar2)
return countries.country_name%type
as
   v_name   countries.country_name%type;
begin
  select country_name
  into v_name
  from countries
  where country_code = p_code;
  return v_name;
end;
```

NO