CS307
Principles of Database Systems

Chapter 15: Summary

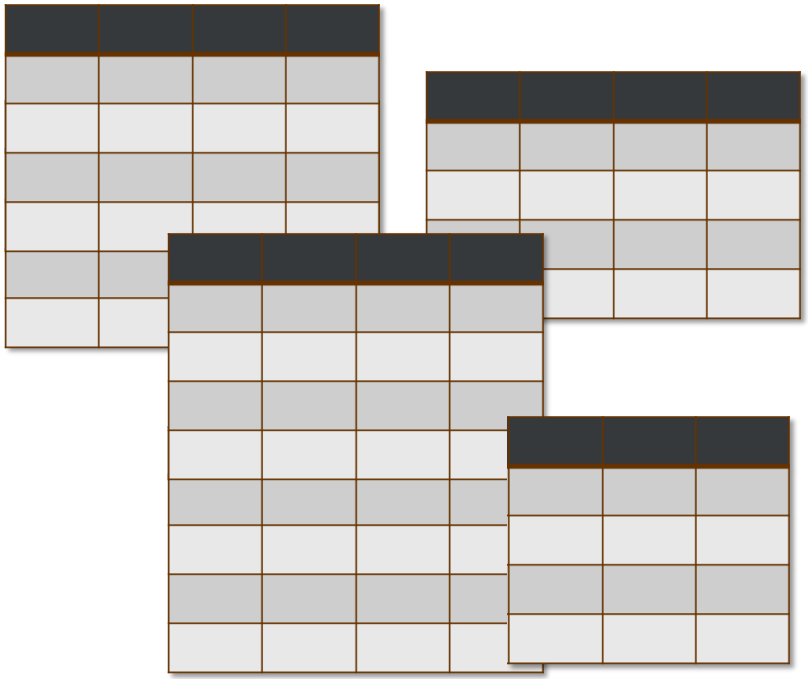
Shiqi YU 于仕琪

yusq@sustech. edu. cn

Relational Database

Based on the relational model of data

- Organizes data into one or more tables
- Rows are also called records or tuples
- Columns are also called attributes



Key

But if you have no duplicates, you need to identify what allows you to differentiate one row from another. It may be one column, or one set of columns, known collectively as a "key".

Flickr: Micky

Normalization

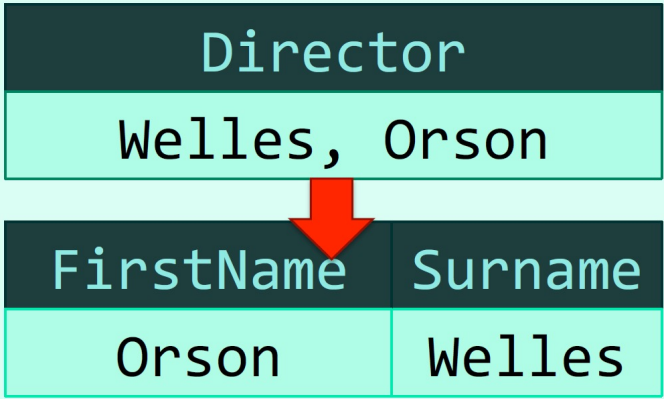
Three simple rules to design a database by Codd in 1971:

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)

Codd, E. F. "Further Normalization of the Data Base Relational Model". IBM Research Report RJ909 (August 31, 1971)

Normalization

Simple attributes



1

Normalization

attributes depend on the full key

Title	Year	Country	Continent
Good Bye, Lenin!	2003	Germany	Europe

2

Wrong. I don't want to repeat it for every German film. It should be said once in a table of countries that Germany is in Europe.

Normalization

non-key attributes not depend on each other

Country	Name	Continent	Currency	Symbol
de	Germany	Europe	Euro	€

3

Wrong. I don't want to repeat it for every country in the Euro zone. I should have it in a currency table, the key of which would be the currency.

Every non key attribute must provide a fact about the key, the whole key, and nothing but the key.

William Kent (1936 – 2005)

William Kent. "A Simple Guide to Five Normal Forms in Relational Database Theory", Communications of the ACM 26 (2), Feb. 1983, pp. 120–125.

SQL

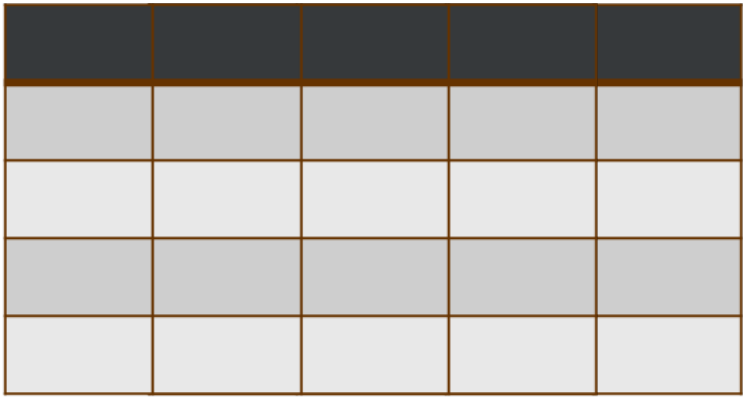
select ...
from ...
where ...

The basic syntax of SQL is very simple. SELECT is followed by the names of the columns you want to return, FROM by the name of the tables that you query, and WHERE by filtering conditions.

CREATE

ALTER

DROP



Data Definition Language

The data definition language (usually called DDL) deals with tables (as well as other database "objects" that we'll see later).

Three commands are enough for creating a new table, changing its structure (for instance adding a new column) or deleting it.

Data Manipulation Language

INSERT

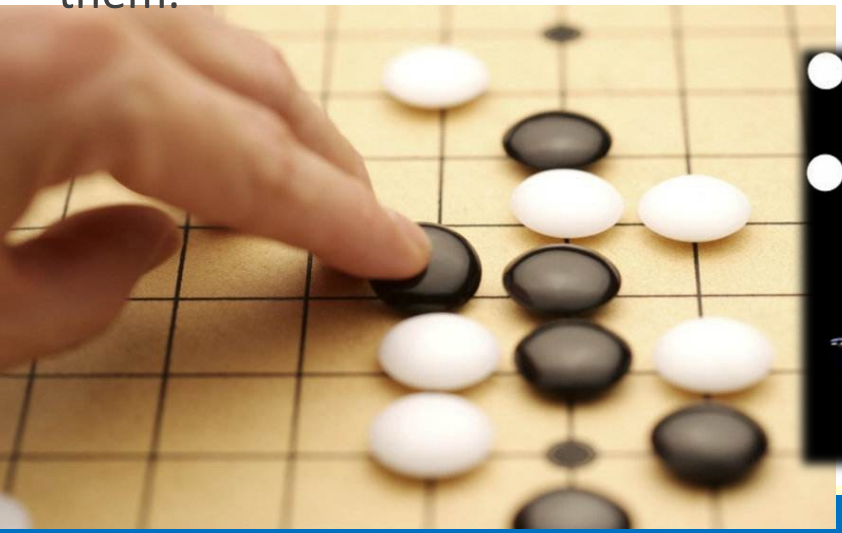
DELETE

UPDATE

SELECT

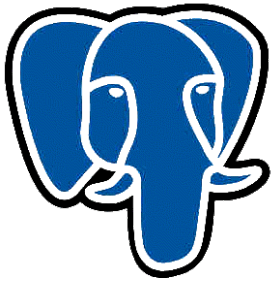
SIMPLE or NOT SIMPLE?

As you can see it's simple. Like chess, it becomes complicated when you COMBINE operations . SQL is one of a few languages where you spend more time thinking about how you are going to do things than actually coding them.



PostgreSQL

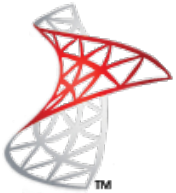
ORACLE®



MySQL®

IBM® DB2®

There is an official SQL standard that no product fully implements, and many subtly and sometimes irritatingly different dialects. We'll see the main variants of SQL.



Microsoft® SQL Server®

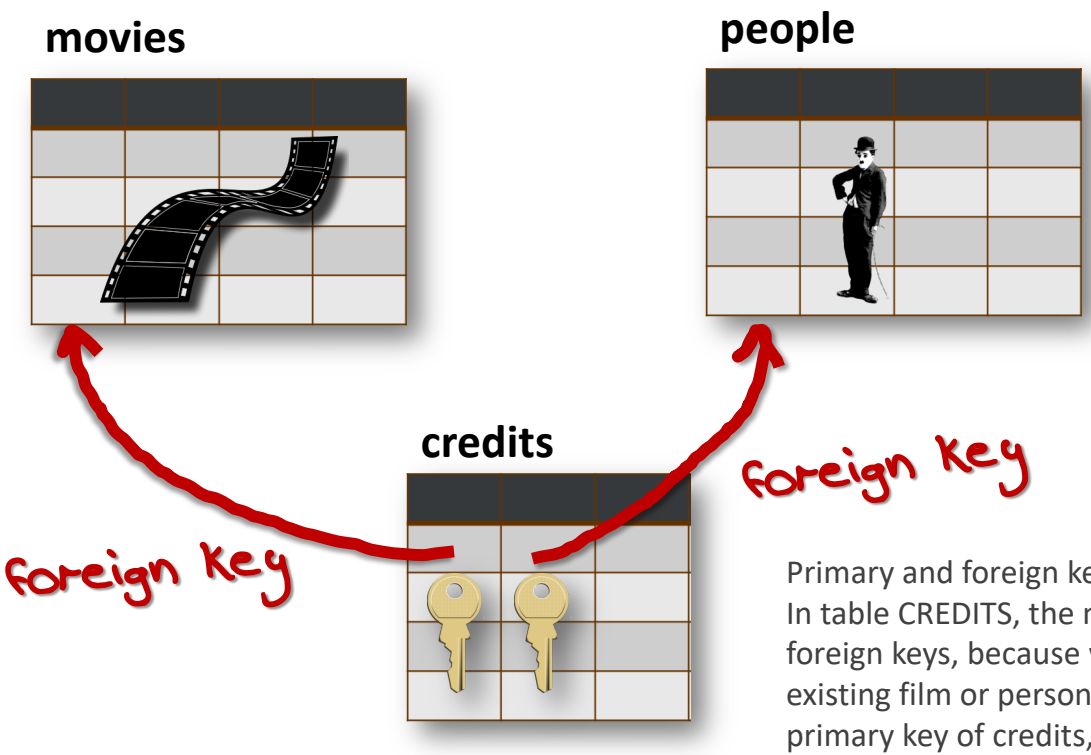
```
CREATE TABLE table_name
(
    ,
    ,
  ...
)
```

This is the syntax (simplified, some products can take a lot of additional options) for creating a table. The weird capitalization here is only to explain that SQL keywords (words that have a special meaning in SQL) are NOT case sensitive and can be typed in any case you want. I mostly use lowercase but some people have different habits.

Reminder

Creating tables requires:

- Proper modelling
- Defining keys
- Determining correct data types
- Defining constraints



Primary and foreign keys are fairly independent notions. In table CREDITS, the movie id and the person id are foreign keys, because we cannot give credits for a non-existing film or person. Both foreign keys are part of the primary key of credits, which is composed of all three columns for this table.

insert into *table_name*
(*list of columns*)
values (*list of values*)

"lists" are comma-separated lists.

Values must match column-names one by one. What happens if you omit a column name from the list? Nothing is entered into it. If the column is mandatory, the INSERT statement fails and nothing at all is done.

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)


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

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.

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?

To delete the row for China in table countries.

The constraint will prevent you to do that.

```
1  delete from countries where country_code='cn';
```

```
[23503] ERROR: update or delete on table "countries" violates foreign key constraint
"movies_country_fkey" on table "movies"
详细: Key (country_code)=(cn) is still referenced from table "movies".
```

select * from *tablename*

To display the full content of a table, you can use select *.

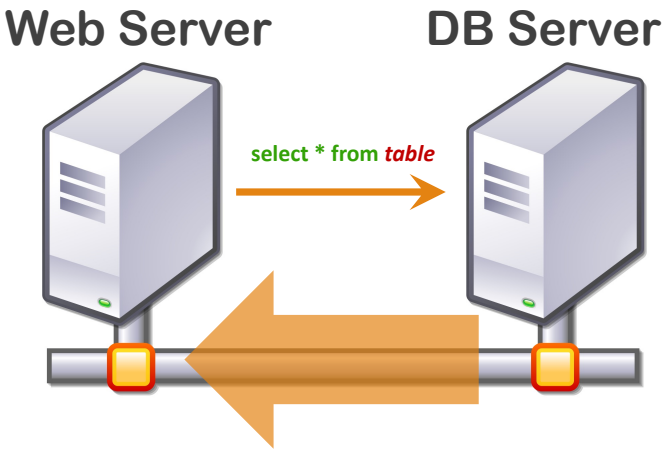
* is short-hand for "all columns" and is frequently used in interactive tools (especially when you don't remember column names ...)

You should not use it, though, in programs.

select * from *table*

≈

print table

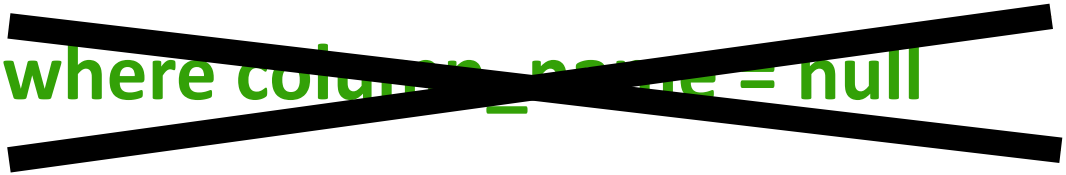


Select * displays the full content of the table and is a bit like printing the table variable.

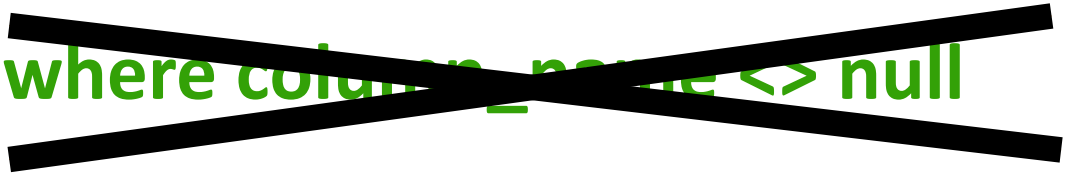
NULL in SQL is NOT a value ...

and if it's not a value, hard to say if a condition is true.

(a lot of people talk about "null values", but they have it wrong)



This for instance, reads "where column name is equal to I don't know what". Even if there is no value (you don’t know it) you cannot say whether something that you don't know is equal to something that you don't know, and it will never be true.



More strangely, perhaps, this reads "where the value in my column is different from I don't know what". If you don't know, you cannot say whether it's different, and so this will also be never true.

The only thing you can test is whether a column contains a value or not, which is done with the special operator IS (NOT) NULL

where column_name is null

where column_name is not null

Arithmetic Operators:

col+NULL -> NULL

col-NULL -> NULL

col*NULL -> NULL

col/NULL -> NULL

…

Remember we have TRUE, FALSE and NULL for logical operations.

Logical operators:

(col > NULL) -> NULL

(col = NULL) -> NULL

…

col is NULL -> True or False

TRUE and NULL -> NULL
FALSE and NULL -> FALSE

TRUE or NULL -> TRUE
FALSE or NULL -> NULL

Throw a NULL in, we have a condition that is never true but because of OR it can just be ignored.

```
col in ('a', 'b', null)
```

=

```
(col = 'a'
 or col = 'b'
 or col = null)
```

If col is 'a', the result is:
TRUE or FALSE or NULL -> TRUE

if col is 'c', the result is:

FALSE or FALSE or NULL -> NULL

if col is NULL, the result is :
NULL or NULL or NULL -> NULL

```
col not in
    ('a', 'b', null)
```

=

```
(col <> 'a'
 and col <> 'b'
 and col <> null)
```

If col is ‘a’, the result is:
FALSE and TRUE and NULL -> FALSE

if col is 'c', the result is:

TRUE and TRUE and NULL -> NULL

if col is NULL, the result is :
NULL and NULL and NULL -> NULL

```
select *
from (select * from movies
        where country = 'us') us_movies
where year_released between 1940 and 1949
```

If it's a valid relation, you can turn it into a "virtual table" by setting it between parentheses and giving a name (us_movies) to the parenthesized expression, and you can apply further filtering to it.

```
where country = 'us'
  or country = 'gb'
 and year_released between 1940
                       and 1949
```

In practice, it means that the SQL engine will process in such a case the **and** before the **or**, and the condition will select British films of the 1940s and all American films irrespective of the date.

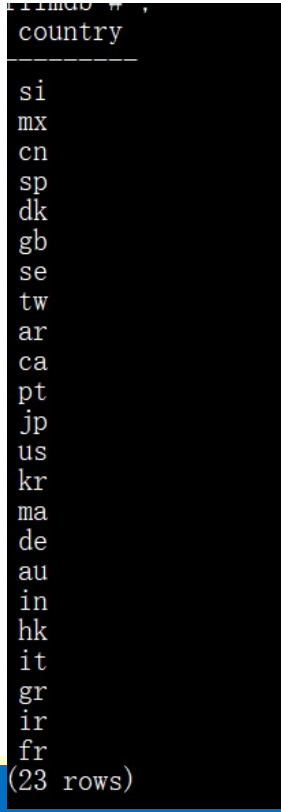
```
select * from movies
where title not like '%A%'
  and title not like '%a%'
```

This expression for instance returns films the title of which doesn't contain any A. This A might be the first or last character as well. Note that if the DBMS is casesensitive you need to cater both for upper and lower case.

If we only are interested in the different countries, there is the special keyword **distinct**.

```
select distinct country from
movies
where year_released=2000
```

The result is a table with one row per country, all of them different, and the only column shown uniquely identifies each row in the result: it's a relation.



```
select distinct country, year_released
from movies
where year_released in (2000,2001)
```

If there are multiple columns after the keywork distinct, distinct will eliminate those rows where all the selected fields are identical.

The selected combination (country, year_released) will be identical.

country	year_released
ar	2000
ar	2001
au	2000
au	2001
br	2001
ca	2000
ca	2001
cn	2000
cn	2001
de	2000
de	2001
dk	2000
fr	2000
fr	2001
gb	2000
gb	2001
gr	2000
hk	2000
hk	2001

Aggregate Functions

Here is a syntax example. We say that we want to group by country, and for each country the aggregate function count(*) says how many movies we have.

group by

```
select country,
       count(*) number_of_movies
from movies
group by country
```

One row for each group

country	number_of_movies
fr	571
ke	1
si	1
eg	11
nz	23
bg	4
ru	153
gh	1
pe	4
hr	1
sg	5
mx	59
cn	200
ee	1
sp	72
cl	14
ec	1
cz	28
dk	30
vn	2
ro	12
mn	1
gb	783
se	59
tw	33
ie	17
ph	42
ar	38
th	21

```
select country,
       year_released,
       count(*) number_of_movies
from movies
group by country,
         year_released
```

You can also group on several columns. Every column that isn't an aggregate function and appears after SELECT must also appear after GROUP BY.

country	year_released	number_of_movies
us	1939	46
nl	2008	1
cn	2016	13
it	1960	10
ch	2011	1
fr	1961	11
us	1931	33
cn	2007	5
mn	2007	1
nz	2010	1
de	1974	2
au	1978	4
us	1935	36
eg	1987	1
in	1937	1
hk	1972	2
is	1996	1
no	2009	1
ru	2010	2
it	1949	5
it	1959	6
gb	2005	13
us	2003	71
ro	2013	1
sp	2008	3
ir	2010	2
jp	1955	3
mx	1974	1
se	1992	2

count(*) / count(col)

min(col)

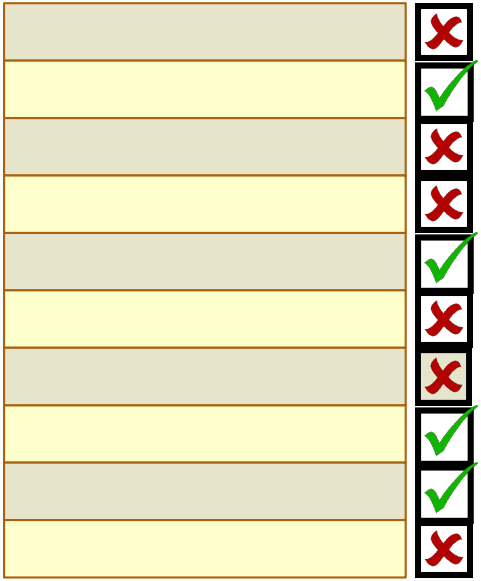
max(col)

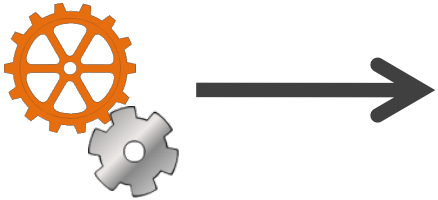
avg(col)

stddev()

These aggregate functions exist in almost all products (SQLite hasn't stddev(), which computes the standard deviation). Most products implement other functions. Some work with any datatype, others only work with numerical columns.

distinct, group by





With a GROUP BY, you must regroup rows before you can aggregate them and return results. In other words, you have a preparatory phase that may take time, even if you return few rows in the end. In interactive applications, end-users don't always understand it well.

There is a short-hand that makes nesting queries unnecessary (in the same way as AND allows multiple filters). You can have a condition on the result of an aggregate with

having

```
select country,
       min(year_released) oldest_movie
from movies
group by country
having min(year_released) < 1940
```

Now, keep in mind that aggregating rows requires sorting them in a way or another, and that sorts are always costly operations that don't scale well (cost increases faster than the number of rows sorted)

```
select title,
       country_name,
       year_released
from movies
    join countries
      on country_code = country
```

The join condition says which values in each table must match for our associating the other columns

movies join countries

1	Casablanca	us 1942	ru	Russia	EUROPE
1	Casablanca	us 1942	us	United States	AMERICA
1	Casablanca	us 1942	in	India	ASIA
1	Casablanca	us 1942	gb	United Kingdom	EUROPE
1	Casablanca	us 1942	fr	France	EUROPE
1	Casablanca	us 1942	cn	China	ASIA
1	Casablanca	us 1942	it	Italy	EUROPE
1	Casablanca	us 1942	ca	Canada	AMERICA
1	Casablanca	us 1942	au	Australia	OCEANIA
2	Goodfellas	us 1990	ru	Russia	EUROPE
2	Goodfellas	us 1990	us	United States	AMERICA
2	Goodfellas	us 1990	in	India	ASIA
2	Goodfellas	us 1990	gb	United Kingdom	EUROPE
2	Goodfellas	us 1990	fr	France	EUROPE
2	Goodfellas	us 1990	cn	China	ASIA
2	Goodfellas	us 1990	it	Italy	EUROPE
2	Goodfellas	us 1990	ca	Canada	AMERICA
2	Goodfellas	us 1990	au	Australia	OCEANIA
3	Bronenosets Potyomkin	ru 1925	ru	Russia	EUROPE
3	Bronenosets Potyomkin	ru 1925	us	United States	AMERICA
3	Bronenosets Potyomkin	ru 1925	in	India	ASIA
3	Bronenosets Potyomkin	ru 1925	gb	United Kingdom	EUROPE
3	Bronenosets Potyomkin	ru 1925	fr	France	EUROPE
3	Bronenosets Potyomkin	ru 1925	cn	China	ASIA
.....			.....		

The join operation will create a virtual table with all combinations between rows in Table1 and rows in Table2.

If Table1 has R1 rows, and Table2 has R2, the huge virtual table has R1xR2 rows.

```
select distinct first_name, surname
from people
    join credits
on credits.peopleid = people.peopleid
where credited_as = 'D'
```

I find it a poor habit to use multiple syntaxes that finally depend on how designers have named their columns, and I prefer using a single syntax that works all the time. If there is some ambiguity, you can remove the ambiguity by prefixing the column name with the table name.

```
select distinct first_name, surname
from people
    join credits
        using (peopleid)
where credited_as = 'D'
```

There is also something called USING (not supported by SQL Server either) which is better and says which commonly named column to use to match rows. However, nothing forces you to have identical names in different tables. In the sample database, the country code is called COUNTRY_CODE in table COUNTRIES, and COUNTRY in table MOVIES. Nothing wrong here.

left outer join

right outer join

full outer join

Books always refer to three kinds of outer joins. Only one is useful and we can forget about anything but the LEFT OUTER JOIN. A right outer join can ALWAYS be rewritten as a left outer join. A full outer join is seldom used.

```
select c.country_name, x.number_of_movies
from countries c
   inner join
   (select country as country_code,
         count(*) as number_of_movies
    from movies
    group by country) x
      on x.country_code = c.country_code
```

We can start by counting in MOVIES how many movies we have per country. This, of course, will only return countries for which there are movies. If we use an inner join, they will be the only ones we'll see.

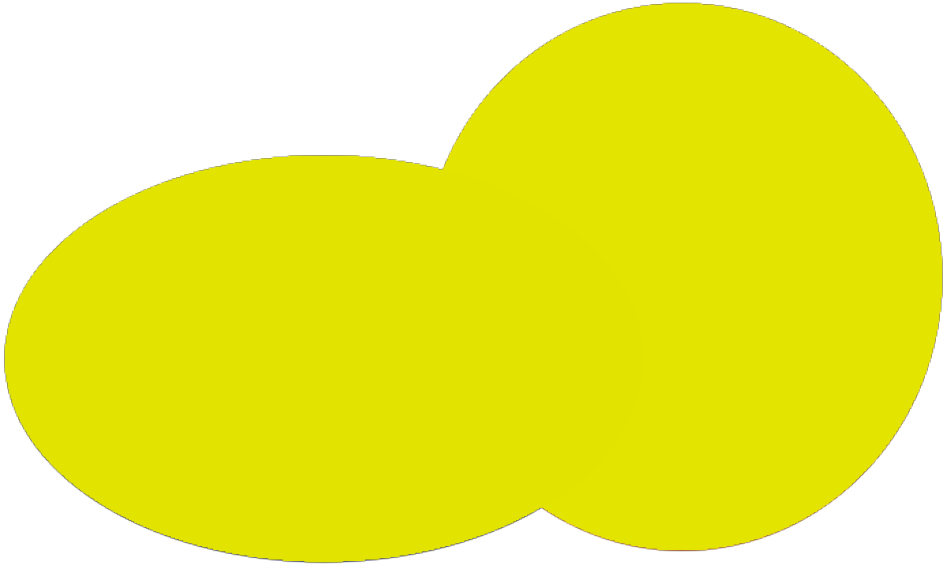
country_name	number_of_movies
Algeria	2
Burkina Faso	2
Egypt	11
Ghana	1
Guinea-Bissau	1
Kenya	1
Libya	2
Mali	2
Morocco	2
Namibia	1
Niger	1
Nigeria	49
Senegal	4
South Africa	10
Tunisia	1
Zimbabwe	1
Argentina	38
Bolivia	4
Brazil	38
Canada	82
Chile	14
Colombia	5
Cuba	4
Ecuador	1
Guatemala	1

```
select c.country_name, x.number_of_movies
from countries c
    left outer join
    (select country as country_code,
          count(*) as number_of_movies
     from movies
     group by country) x
        on x.country_code = c.country_code
```

With a left outer join, we'll see all countries in the COUNTRIES table appear. Note that the table that we want to see listed in full (COUNTRIES in that case) is always with a LEFT OUTER JOIN the first one after FROM.

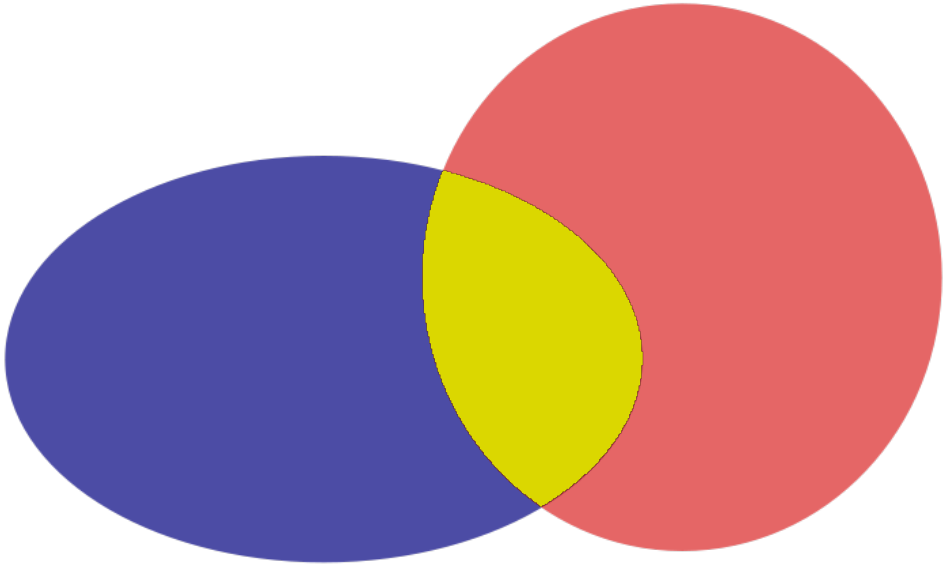
country_name	number_of_movies
Algeria	2
Angola	
Benin	
Botswana	
Burkina Faso	2
Burundi	
Cameroon	
Central African Republic	
Chad	
Comoros	
Congo Brazzaville	
Congo Kinshasa	
Cote d'Ivoire	
Djibouti	
Egypt	11
Equatorial Guinea	
Eritrea	
Ethiopia	
Gabon	
Gambia	
Ghana	1
Guinea	
Guinea-Bissau	1
Kenya	1
Lesotho	
Liberia	
Libya	2

union



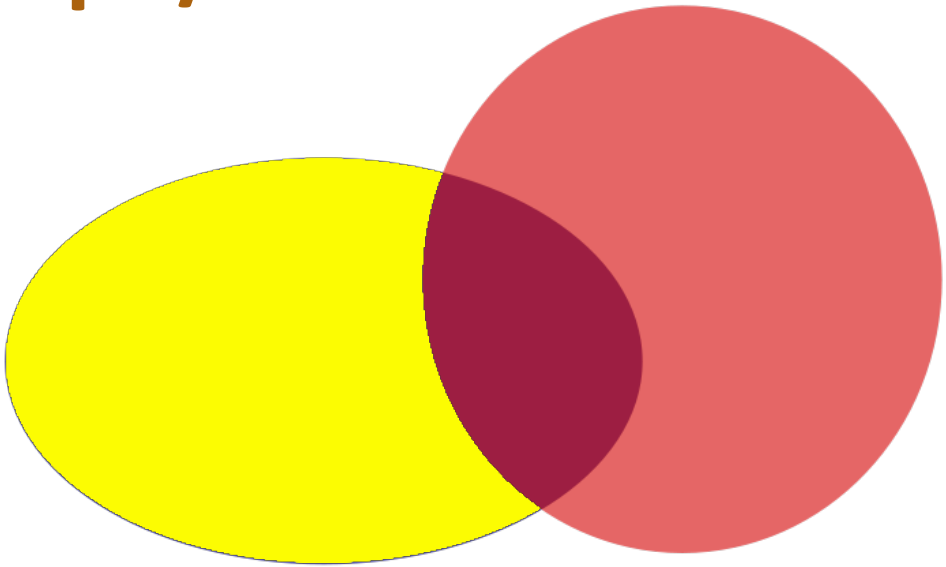
UNION is the most used set operator. It's not the only one.

intersect



It returns the common rows in two tables (or query results)

except / minus



EXCEPT, called MINUS in Oracle, is the last one. It returns the rows from the first table, minus those that can also be found in the second table.

```
select title, year_released
from movies
where country = 'us'
order by year_released
```

We can apply it to any result set ...

```
order by col1 desc, col2 asc, ...
```

You can specify that a sort is descending by following the column name with DESC. You can also use ASC to say ascending, but as it's the default nobody uses it.

Another very important (but not available in MySQL before 8 or SQLite) set of functions for ordering/reporting are window functions. They bear different names, Oracle calls them analytic functions, DB2 calls them OLAP (OnLine Analytical Processing) functions. They are of two kinds, we'll start with non-ranking functions.

Non-ranking

Window Functions

Ranking

Window functions hold the middle-ground between scalar and aggregate functions. Like scalar functions, they return a result for a single row; but like aggregate functions, this result is computed out of several rows. The syntax is as follows

func(parameters) over (magic clause)

With DBMS products that support window functions, every aggregate function can be used as a window function. Instead of specifying with GROUP BY the subset on which the result is computed, you say OVER (PARTITION BY ...)

```
min(year_released)
  over (partition by country)
```

```
select country,
       title,
       year_released,
       min(year_released)
       over (partition by country)
              earliest_year
from movies
```

Thus, this query returns two years for every movie: the one when this particular movie was released, and the one when the earliest movie for the same country was released. You get both detail and an aggregate value on the same row.

TITLE and year of the earliest movie per country

country	title	year_released	earliest_year
am	Sayat Nova	1969	1969
ar	La Ciénaga	2001	1945
ar	La bestia debe morir	1952	1945
ar	Truman	2015	1945
ar	Waiting for the Hearse	1985	1945
ar	El hombre de al lado	2010	1945
ar	Derecho de familia	2006	1945
ar	Carancho	2010	1945
ar	Savage Pampas	1966	1945
ar	Cama adentro	2004	1945
ar	Un cuento chino	2011	1945
ar	El hijo de la novia	2001	1945
ar	Delirium	2014	1945
ar	Madame Bovary	1947	1945
ar	La hora de los hornos	1968	1945
ar	El abrazo partido	2004	1945
ar	Hombre mirando al sudeste	1986	1945
ar	Crónica de una fuga	2006	1945
ar	Las aventuras del Capitán Piluso	1963	1945
ar	Albéniz	1947	1945

There are three main ranking functions. In many cases, they return identical values. Differences are interesting.

row_number()

rank()

dense_rank()

With a ranking window function you MUST have an ORDER BY clause in the OVER() (you cannot have an empty OVER() clause). You can combine it with a PARTITION BY to order with groups.

```
over (order by ...)

over (partition by ...
      order by ...)
```

This idea that one business operation may translate into several database operations that must all succeed or fail is of prime importance to a DBMS. Some products require a special command to start a transaction (BEGIN is sometimes START)

```
begin transaction

    insert
      update
        delete
```

Other products such as Oracle or DB2 automatically start a transaction if you aren't already in one when you start modifying data.

A transaction ends when you issue either COMMIT (which is like an OK button) or ROLLBACK, which cancels everything you have done since the beginning of a transaction (you can sometimes cancel a subpart of a transaction, but it's not much used)

commit

ROLLBACK automatically undoes everything. You can no longer do it after COMMIT.

rollback

Functions

Build-in functions:
lower()
upper()
substring()
trim()
…

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.

Here is a PostgreSQL example


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

SQL FIRST!

Tom Kyte, who is Senior Technology Architect at Oracle, says that his mantra is:

- You should do it in a single SQL statement if at all possible.
- If you cannot do it in a single SQL statement, then do it in PL/SQL (as little PL/SQL as possible!)

What Prof. Yu says:

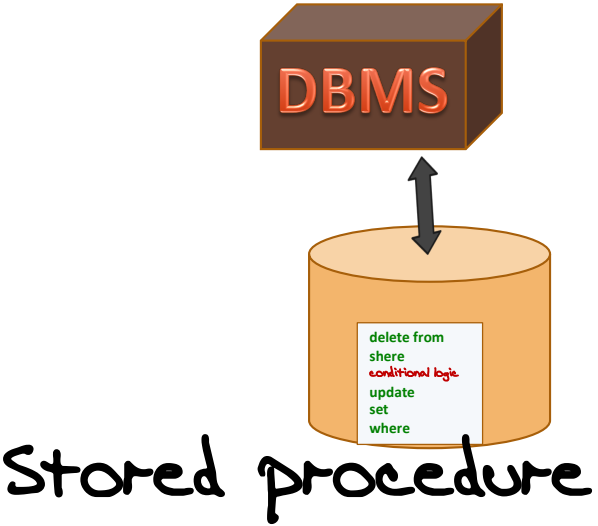
- You should ask for help from someone more experienced than you, Google, forums, etc.

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)

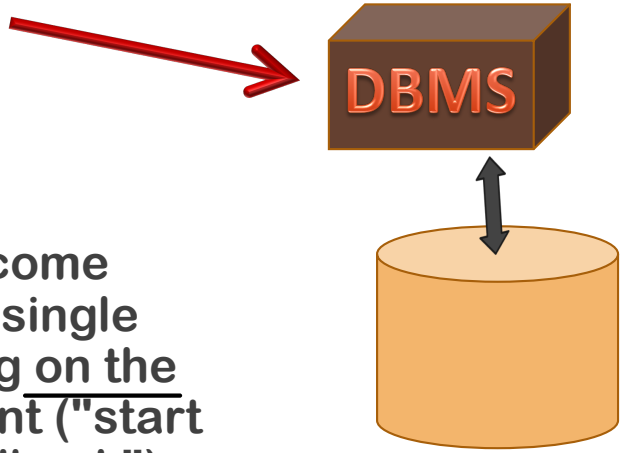
create procedure myproc

```
delete from ...
where  ...
+ Conditional logic
update ...
set ...
where  ...
```



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

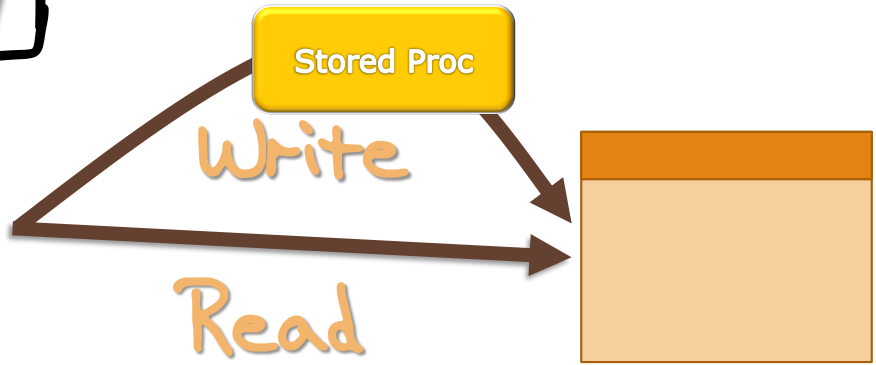
execute myproc



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.

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



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.

Time

PostgreSQL

ORACLE

IBM DB2

MySQL

SQLite

Microsoft 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

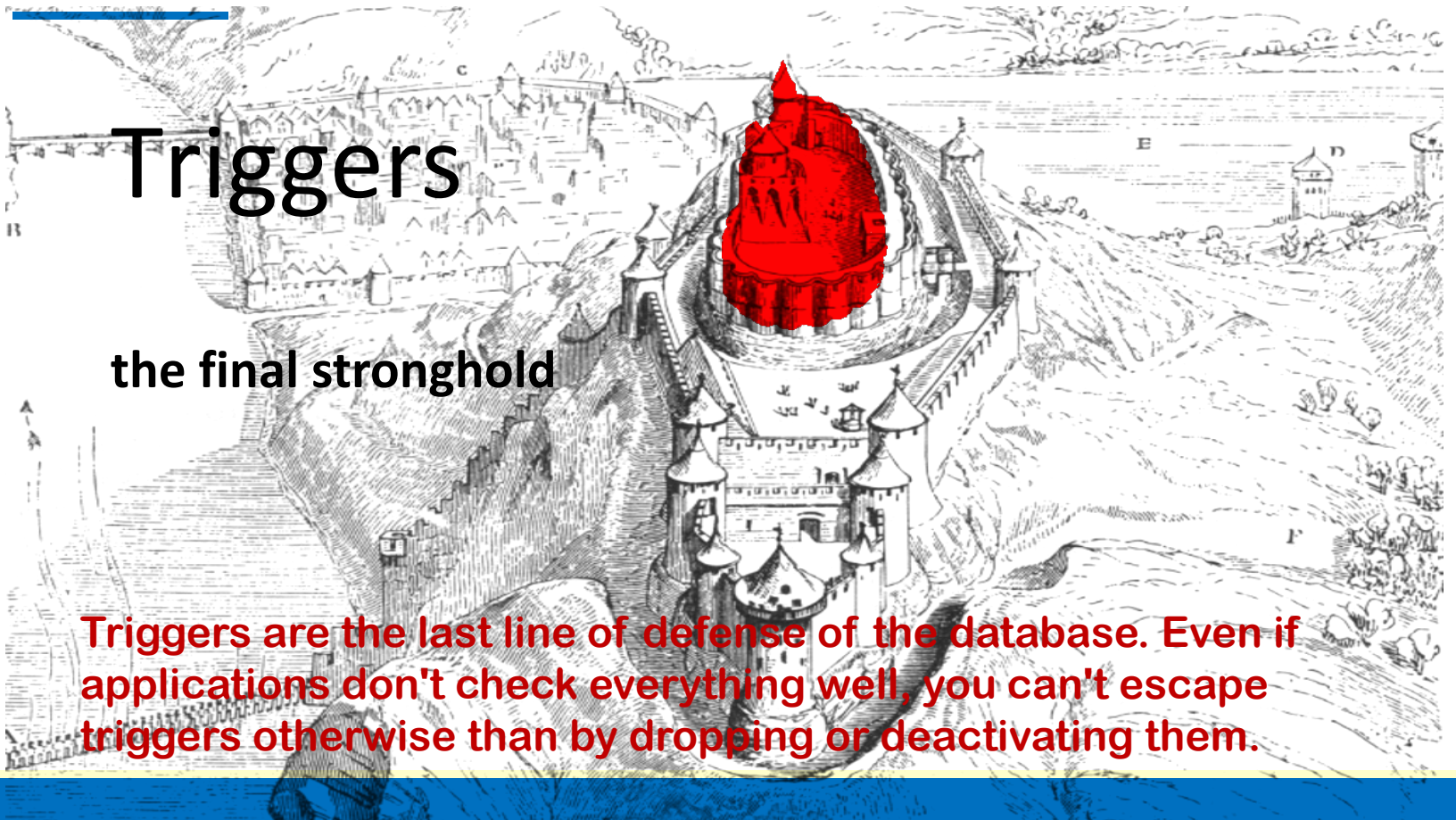
old table
new table

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

```
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)

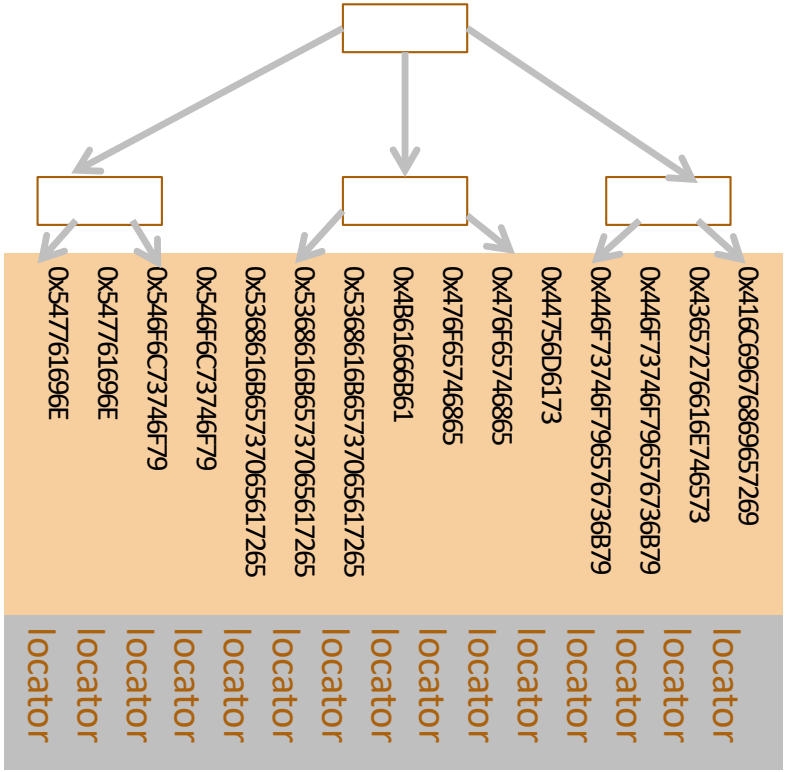
Triggers



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.

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 giving it a name and specifying table name 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.

```
create table movies
(
    movieid       integer      not null
        constraint movies_pkey
            primary key,
    title         varchar(100) not null
        constraint "title length"
            check (length((title)::text) <= 100),
    country       char(2)      not null
        constraint movies_country_fkey
            references countries
        constraint "country length"
            check (length(country) <= 2),
    year_released integer      not null
        constraint "year_released numerical"
            check ((year_released + 0) = year_released),
    runtime       integer
        constraint "runtime numerical"
            check ((runtime + 0) = runtime),
    constraint movies_title_country_year_released_key
        unique (title, country, year_released)
);
```

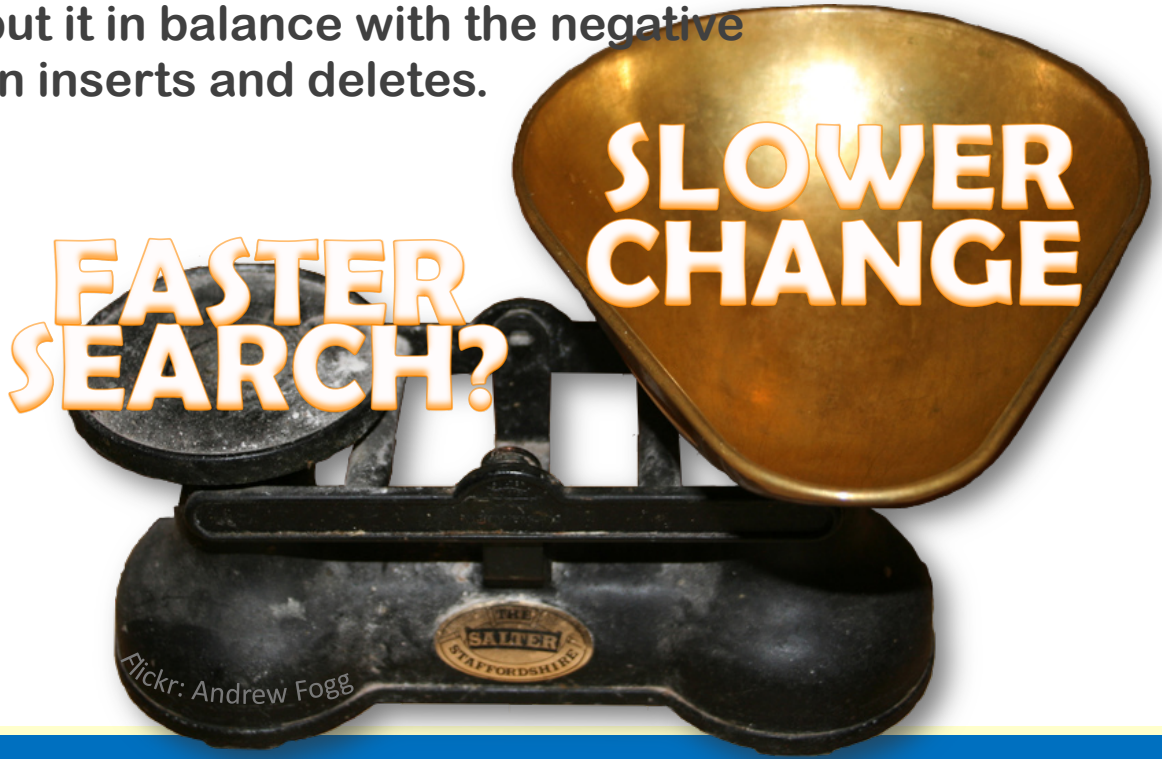
Primary key

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.

You always have to compromise. Even if an index makes a search significantly faster, you have to put it in balance with the negative impact on inserts and deletes.



What is the DBMS actually DOING?

If there is an index on a column such as COUNTRY, will the DBMS use it? Not necessarily. The optimizer may decide not to use an index. Is there a way to know if it will? Good news:

yes.


```
movies=# explain select distinct m.title, m.year_released
movies-# from movies m
movies-#      inner join credits c
movies-#        on c.movieid = m.movieid
movies-#        inner join people p
movies-#          on p.peopleid = c.peopleid
movies-# where p.surname = 'Bogart';
                              QUERY PLAN
-----------------------------------------------------------------------------------------
 HashAggregate  (cost=5.29..5.30 rows=1 width=222)
   ->  Nested Loop  (cost=2.16..5.28 rows=1 width=222)
         ->  Hash Join  (cost=2.16..4.51 rows=1 width=4)
               Hash Cond: (c.peopleid = p.peopleid)
               ->  Seq Scan on credits c  (cost=0.00..1.97 rows=97 width=8)
               ->  Hash  (cost=2.15..2.15 rows=1 width=4)
                     ->  Seq Scan on people p  (cost=0.00..2.15 rows=1 width=4)
                           Filter: ((surname)::text = 'Bogart'::text)
         ->  Index Scan using movies_pkey on movies m  (cost=0.00..0.76 rows=1 width=226)
               Index Cond: (movieid = c.movieid)
(10 rows)

movies=#
```

```
create view viewname      (col1,…,coln)
as
select ...
```

In practice (theory is a bit more complicated) there isn't much to a view: it's basically a named query. If the query is correct, it should return a valid relation, so why not consider it as if it were a table? You can optionally rename columns after the view name (if you don't, the view uses column names from the query result)

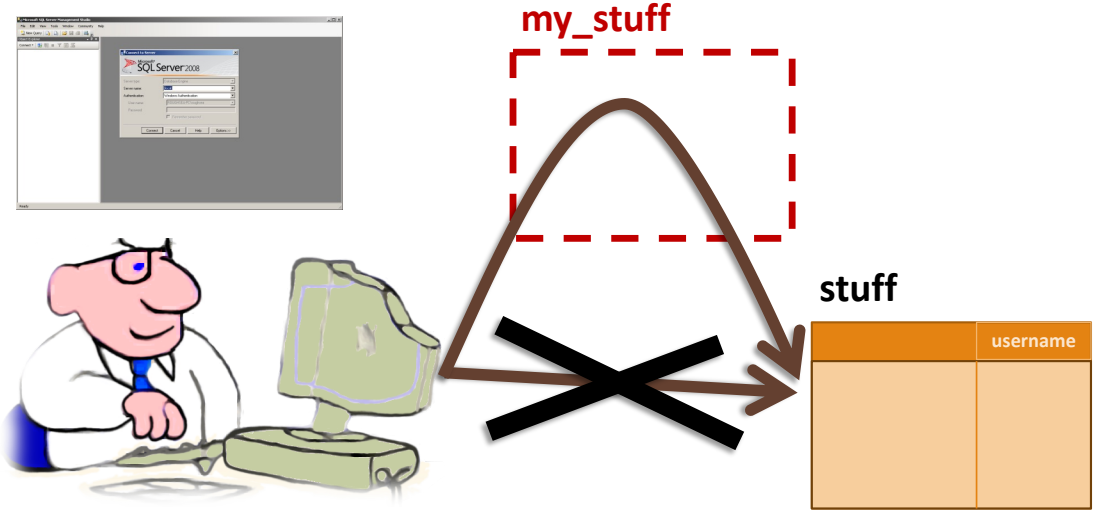
VIEWS

Hide complexity

The problem with views is that as long as you haven't seen how they have been defined, you have no idea how complex they may be. They may be fairly innocuous, or they may be queries of death (they often are)

DON'T

create views on complex views



Now, the problem is that for security though a view, users need to be personally authenticated.

What about **CHANGING DATA** through views?

If views are in theory like tables, why not using them for controlling not only what you SEE, but what you CHANGE?

Lots of things can go wrong

It all depends on the view ... The problem is that most view are designed to provide a more user-friendly view of data: joins transforming codes into more legible values, functions making data prettier (date formatting, for instance). And by doing so you often lose information.

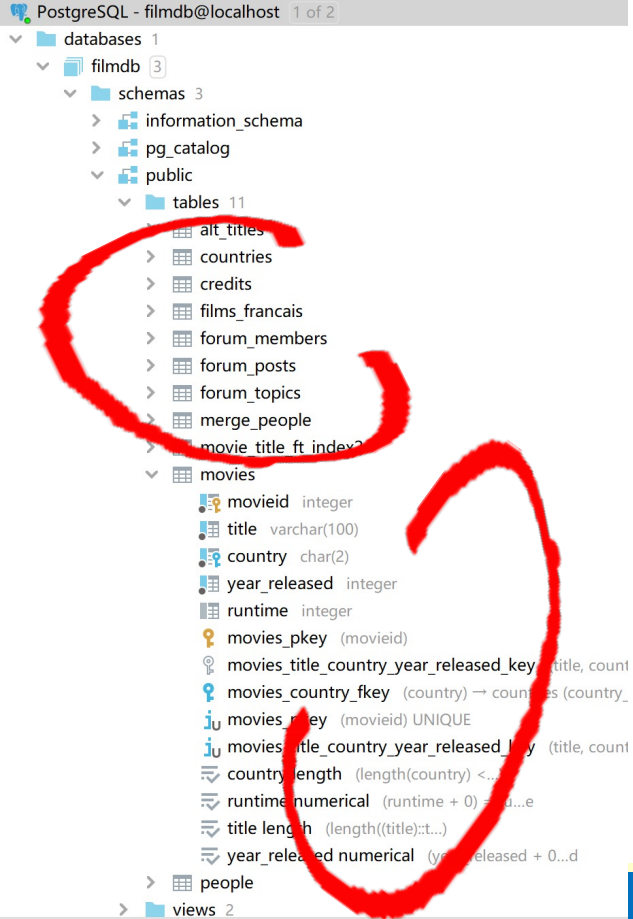
One very good example of view application is the set of tables that contain information about the objects in the database, collectively known as the

Data Dictionary

or sometime called the

Catalog

They are using all the features we have seen (you only have privileges to read views and only see what is relevant to your account)



Any database stores "metadata" that describes the tables in your database (and not only them)

All client tools use this information to let you browse the structure of your tables (here it's SQL Server, Visual Studio)

```
select m.title, m.year_released
from movies m
  inner join credits c
  on c.movieid = m.movieid
  inner join people p
  on p.peopleid = c.peopleid
where p.first_name = 'Tim'
 and p.surname = 'Burton'
 and c.credited_as = 'D'
```

Syntax ✓

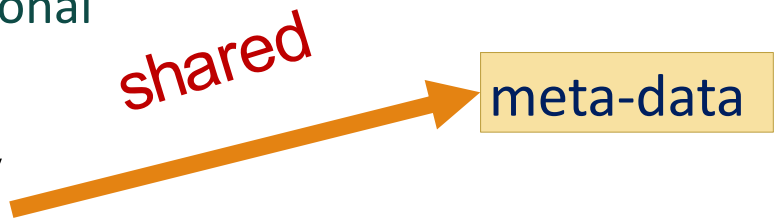
Do tables exist? ✓

Right to access? ✓

Do columns exist? ✓

Indexes we can use? Best way to access data? ✓

One way to improve efficiency is to keep data dictionary information (meta-data) in a shared cache to avoid additional queries.



Kept in memory

```
select m.title, m.year_released
from movies m
  inner join credits c
  on c.movieid = m.movieid
  inner join people p
  on p.peopleid = c.peopleid
where p.first_name = 'Tim'
 and p.surname = 'Burton'
 and c.credited_as = 'D'
```

Syntax ✓

Do tables exist? ✓

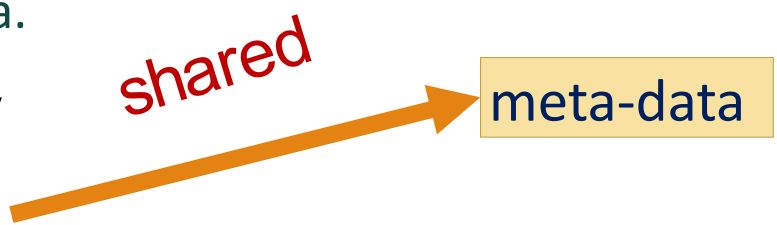
Right to access? ✓

Do columns exist? ✓

Indexes we can use? Best way to access data? ✓

Another crucial phase is the one when the optimizer tries to determine the most efficient way to access data.

Kept in memory



SCALING UP

For many years, the answer to a database outgrowing the processing power of its server has been to replace the server by a bigger server.

SCALING OUT

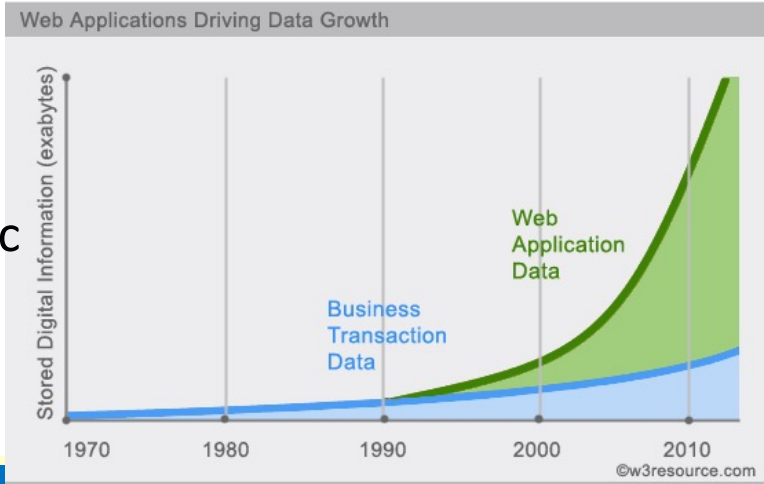
This is why people quickly thought of an alternative, adding more servers and making them share the load.

What is NoSQL?

- Stands for “NOT ONLY SQL”
- A **Non-Relational** database (No Tables)
- A flexible database used for **big data** & **real-time** web apps
- Multiple types of NoSQL databases

What is Big Data?

- A term for data sets that are so large that traditional methods of storage & processing are inadequate.
- Massive increase in data volume within the last decade or so
- Social networks, search engines, etc
- Challenges in storage, capture, analysis, transfer, etc



https://www.runoob.com/mongodb/nosql.html

Advantages of NoSQL over RDBMS

- Handles Big Data
- Data Models – No predefined schema
- Data Structure – NoSQL handles unstructured data
- Cheaper to manage
- Scaling – Scale out / horizontal scaling

Advantages of RDBMS over NoSQL

- Better for relational data
- Normalization
- Well known language (SQL)
- Data Integrity
- ACID Compliance (atomicity, consistency, isolation, durability)