CS307

Database Principles

Stéphane Faroult
faroult@sustc.edu.cn

Liu Zijian liuzijian47@163.com

Our example database

Film database
Available online at
`http://edu.konagora.com/SQLsandbox.php`

Bigger database available as a sqlite file in Sakai
`http://sqlite.org/`
`http://sqlitebrowser.org/`
This one can access a lot of databases but is hard to set up → `http://www.squirrelsql.org/`

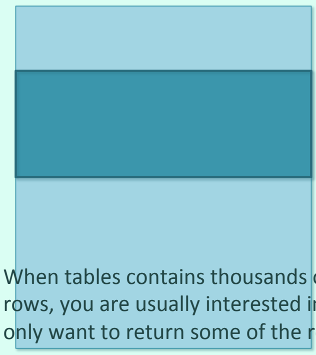
select * from *table*

≈

print table

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

Restriction



When tables contains thousands or millions or billions of rows, you are usually interested in only a small subset, and only want to return some of the rows.

Filtering is performed in the "where" clause, with conditions that are usually expressed by a column name followed by a comparison operator and the value to which the content of the column is compared.

```
select * from movies
where country = 'us'
```

Only rows for which the condition is true will be returned.

number

'constant'

column

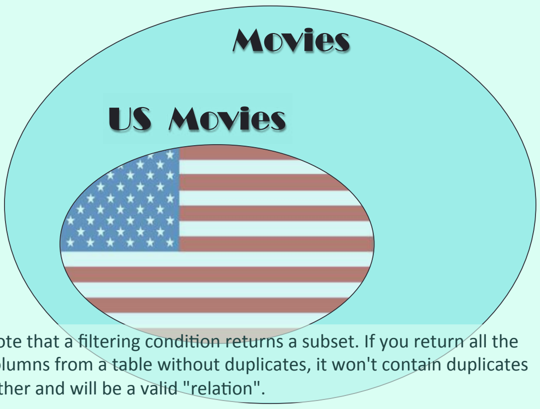
You can compare to a number, a string constant, another column (from the same table or another, we'll see queries involving several tables later) or the result of a function (we'll see them soon)

Column doesn't exist!

```
where title =  Jaws

where title = 'Jaws'
```

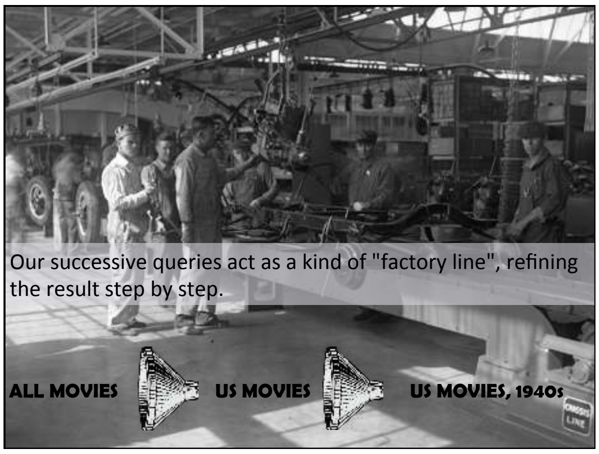
Beware that string constants must be quoted between single-quotes. If they aren't quoted, they will be interpreted as column names. Same thing with Oracle if they are double-quoted.



Note that a filtering condition returns a subset. If you return all the columns from a table without duplicates, it won't contain duplicates either and will be a valid "relation".

```
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.


Our successive queries act as a kind of "factory line", refining the result step by step.

```
select *
from (select * from movies
      where year_released between 1940
                                  and 1949)
                   movies_from_the_1940s
where country = 'us'
```

Note that it would be functionally equivalent (although not necessarily performance-wise) to start by retrieving films from the 1940s for all countries, then filtering the American ones.

On a single table query, nobody would ever do that and what we use are simply "and" conjunctions. However, we shall soon see more complex queries for which thinking by successive steps is critical.

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

or **not**

You can also link conditions with OR and negate them with NOT. Beware of boolean logic: in a program, an IF or WHILE rarely contains more than 2 or 3 conditions. An SQL query can contain dozens, and boolean logic can become quite complicated.

One thing to remember is that, like numerical operators, all logical operators haven't the same precedence and, like * is "stronger" than +, and is "stronger" than or.

and > or

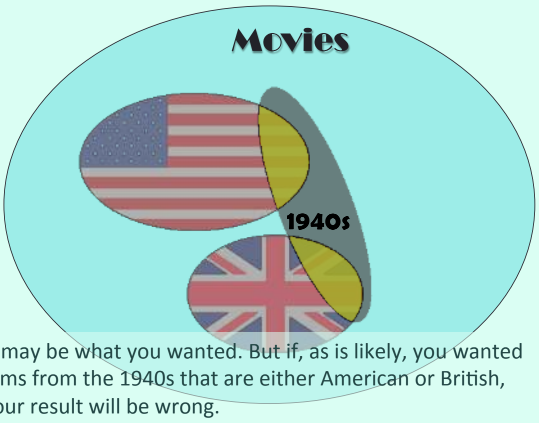
$$\underbrace{2 + \underbrace{3 * 4}_{}}_{}$$

$$\underbrace{2 + \quad 12}_{}$$

$$14$$

```
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.

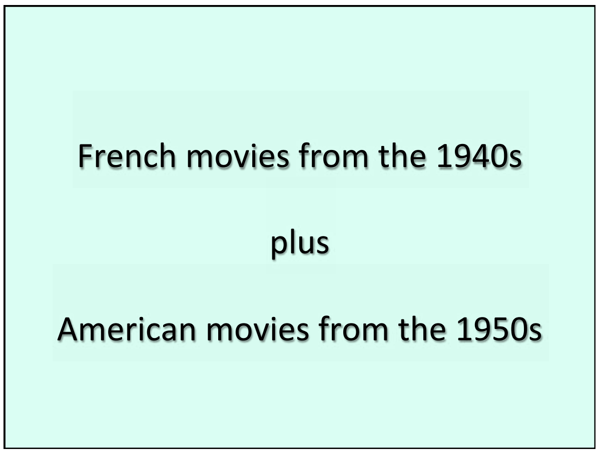


It may be what you wanted. But if, as is likely, you wanted films from the 1940s that are either American or British, your result will be wrong.

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

What you should do is do in such a case in an arithmetical expression: use parentheses to specify that the or should be evaluated before the and, and that the conditions filter

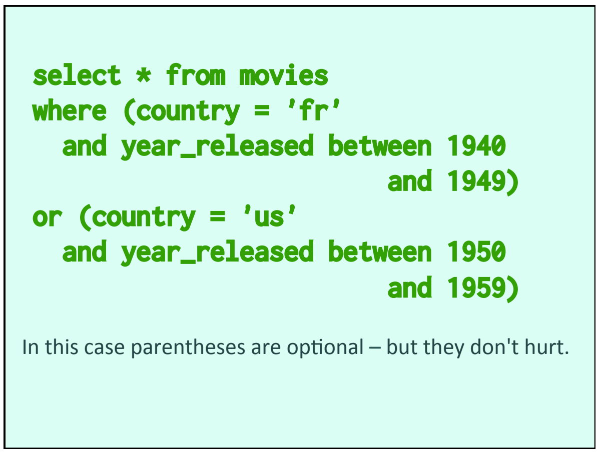
1) British or American films
2) That were released in the 1940s



French movies from the 1940s

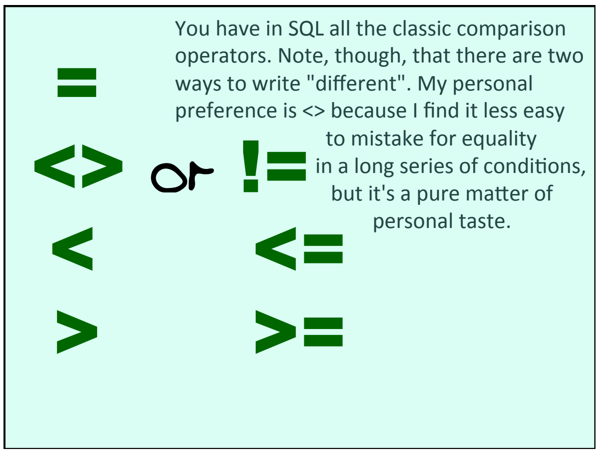
plus

American movies from the 1950s



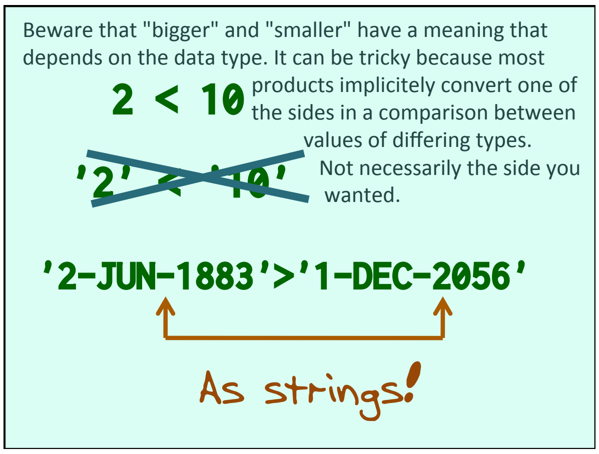
```
select * from movies
where (country = 'fr'
  and year_released between 1940
                        and 1949)
or (country = 'us'
  and year_released between 1950
                        and 1959)
```

In this case parentheses are optional – but they don't hurt.



```
=
<>  or  !=
<       <=
>       >=
```

You have in SQL all the classic comparison operators. Note, though, that there are two ways to write "different". My personal preference is <> because I find it less easy to mistake for equality in a long series of conditions, but it's a pure matter of personal taste.



Beware that "bigger" and "smaller" have a meaning that depends on the data type. It can be tricky because most products implicitely convert one of the sides in a comparison between values of differing types. Not necessarily the side you wanted.

```
2 < 10
```

~~'2' < '10'~~

```
'2-JUN-1883'>'1-DEC-2056'
```

As strings!

Beware also of date formats, and of conflicting European/ American formats which can for some dates be ambiguous. Common problem in multinational companies.

DD/MM/YYYY

MM/DD/YYYY

The conclusion is that whenever you are comparing data of slightly different types, you should use functions (they don't always bear the same names but exist with all products) that "cast" data types. It will avoid bad surprises.

Convert EXPLICITLY!

Another frequent mistake is with datetime values. If you compare them to a date (without any time component) the SQL engine will not understand that the date part of the datetime should be equal to that date.

```
where issued = <some date>
```

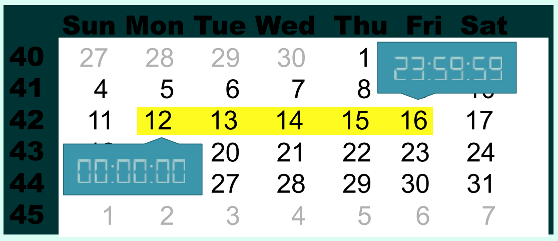
Flickr:Yoppy & Rudolf Schuba

Rather, it will consider that the date that you have supplied is actually a datetime, with the time component that you can read below.

Everything that won't have happened at that precise second won't satisfy the condition and won't be returned.

An equality condition on a single datetime might return no or very few rows, and alert you. You may not as easily notice with the following condition that you are going to miss all the rows that were issued on Friday, which probably wasn't the intent.

```
 where issued >= <Monday's date>
                  <Monday 00:00:00>
   and issued <= <Friday's date>
                  <Friday 00:00:00>
```



	Sun	Mon	Tue	Wed	Thu	Fri	Sat
40	27	28	29	30	1		
41	4	5	6	7	8		
42	11	12	13	14	15	16	17
43			20	21	22	23	24
44			27	28	29	30	31
45	1	2	3	4	5	6	7

A correct condition would select everything from Monday at 00:00:00 to Friday at 23:59:59 included, or Saturday at 00:00:00 excluded.

There are also a few comparison operators that are specific to SQL. You have seen BETWEEN

```
year_released between 1940 and 1949
```

It's shorthand for this:

```
    year_released >= 1940
and year_released <= 1949
```

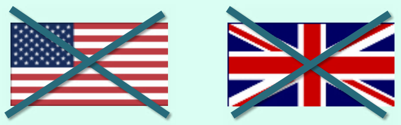
Beware that it's a very dumb replacement, and that the order is important : BETWEEN 1949 AND 1940 might look equivalent to you but would return no rows.

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

```
where country in ('us', 'gb')
  and year_released between 1940 and 1949
```

Another interesting operator is IN () which can be used as the equivalent for a series of equalities with OR (it has also other interesting uses). It may make a comparison clearer than a parenthesized expression.

```
country not in ('us', 'gb')
```



Of course all comparisons can be negated with NOT.

like

% **_**

For strings, you also have LIKE which is a kind of regex (regular expression) for dummies. LIKE compares a string to a pattern that can contain two wildcard characters, % meaning "any number of characters, including none" and _ meaning "one and only one character"

```
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 case-sensitive you need to cater both for upper and lower case.

Another way to handle case is to force it, but performance-wise it's a terrible idea as we'll see later.

```
select * from movies
where upper(title)
            not like '%A%'
```

Not good to apply a function to a searched column

Note that most products also have "true" regex functions and expressions, but for similar performance concerns they are better used to refine an already narrowed-down result.

There are also expressions for this particular SQL animal, NULL

In a language such as C, you can compare a pointer to NULL, because NULL is defined as the '0' pointer. In Java, you can also compare an object to null if it hasn't been created yet. Not in SQL, where NULL denotes that a value is missing.

```
if (ptr == NULL) {
  ...
```

Not the same as NULL in SQL!!

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)

```
~~where column_name = null~~
```

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.

~~where column_name <> null~~

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.

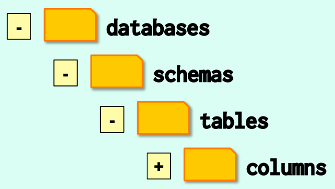
The only thing you can test is whether a column contains a value or not, which is done with the special operator IS (NOT) NULL

```
where column_name is null

where column_name is not null
```

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

You can also, instead of selecting '*', select only some columns. Of course, it requires knowing what are the columns in the table and how they are named.



Graphical tools usually have some kind of hierarchy allowing you to access the description of a table. I'll explain the hierarchy in some detail later.

```
desc movies;
```

```
describe table movies
```

```
\d movies
```

```
.schema movies
```

Command lines tools also all have a command. As usual, most commands are different.

One important feature of SQL is that you needn't return data exactly as it was stored. Operators, and a large number of (mostly DBMS specific) functions allow to return transformed data.

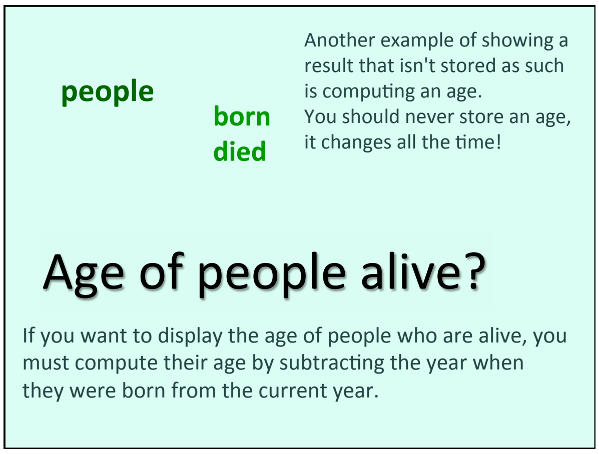
A simple transformation is concatenating two strings together. Most products use || (two vertical bars) to indicate string concatenation. SQL Server, though, uses +, and MySQL a special concat() function that also exists in some other products.

```
'hello'|| ' world'
'hello' + ' world'
concat('hello' , ' world' )
```

Note that you can give a name to an expression. This will be used as column header. It also becomes a "virtual column" if you turn the query into a "virtual table".

```
select title
        || ' was released in '
        || year_released movie_release
from movies
where country = 'us'
```

Although YEAR_RELEASED is actually a number, it's implicitly turned into a string by the DBMS. In that case it's not a big issue but it would be better to use a function to convert explicitly.



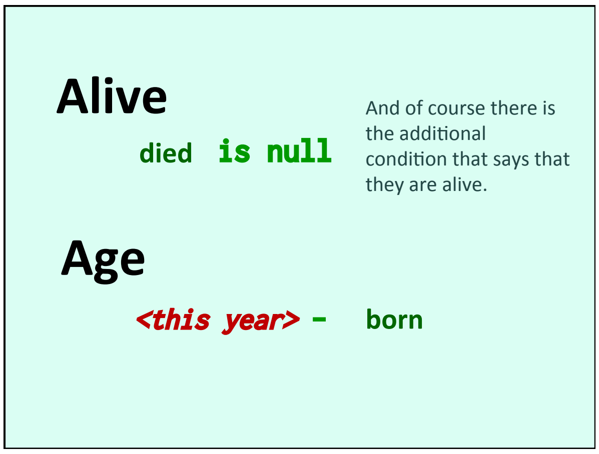
people

born
died

Another example of showing a result that isn't stored as such is computing an age.
You should never store an age, it changes all the time!

Age of people alive?

If you want to display the age of people who are alive, you must compute their age by subtracting the year when they were born from the current year.



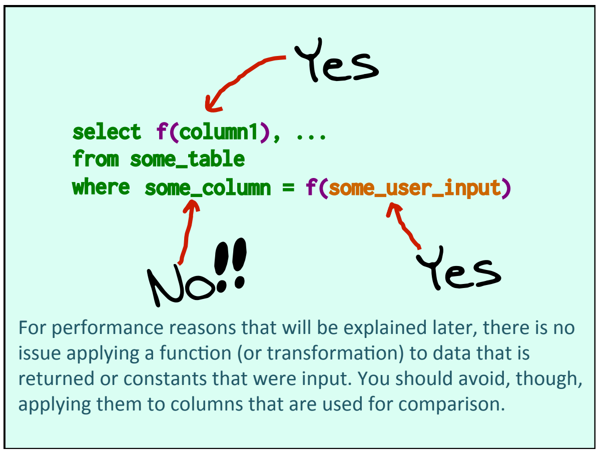
Alive

died **is null**

And of course there is the additional condition that says that they are alive.

Age

<this year> - born

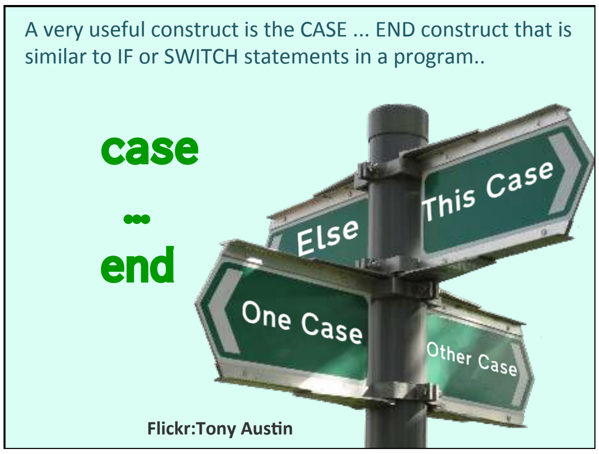


Yes

```
select f(column1), ...
from some_table
where some_column = f(some_user_input)
```

No!! Yes

For performance reasons that will be explained later, there is no issue applying a function (or transformation) to data that is returned or constants that were input. You should avoid, though, applying them to columns that are used for comparison.



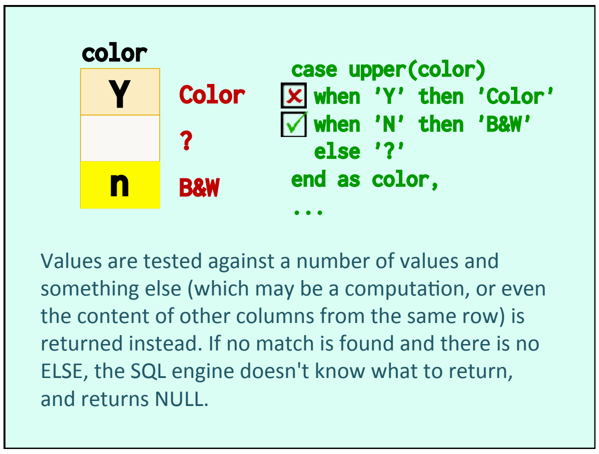
A very useful construct is the CASE ... END construct that is similar to IF or SWITCH statements in a program..

case

...

end

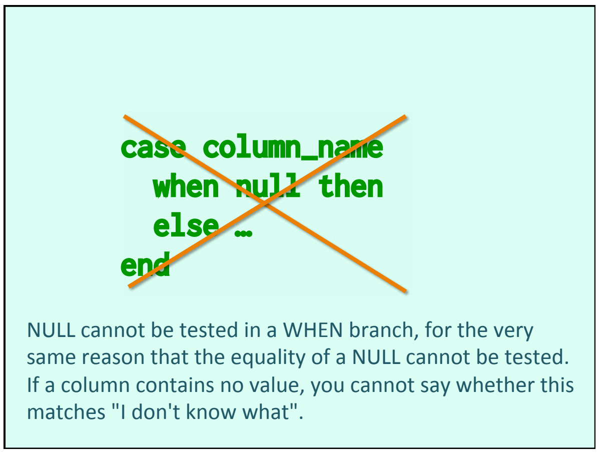
Flickr:Tony Austin



color	
Y	Color
	?
n	B&W

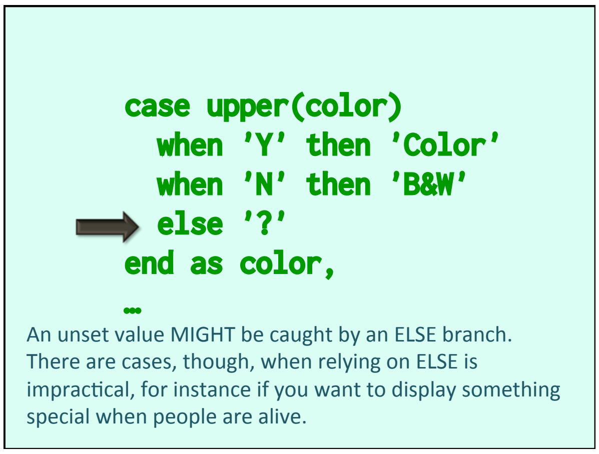
```
  case upper(color)
☒ when 'Y' then 'Color'
☑ when 'N' then 'B&W'
    else '?'
  end as color,
  ...
```

Values are tested against a number of values and something else (which may be a computation, or even the content of other columns from the same row) is returned instead. If no match is found and there is no ELSE, the SQL engine doesn't know what to return, and returns NULL.



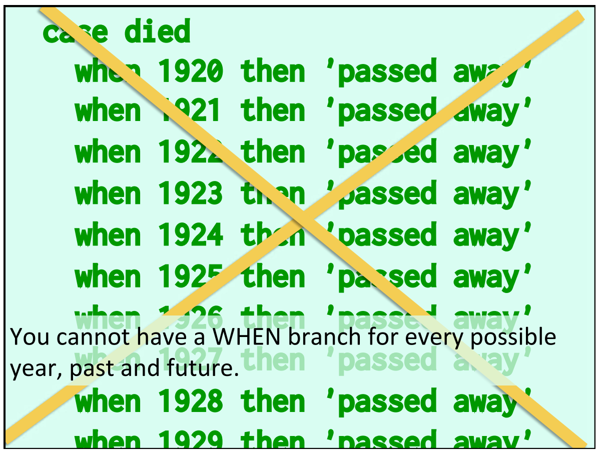
```
~~case column_name~~
~~  when null then~~
~~  else …~~
~~end~~
```

NULL cannot be tested in a WHEN branch, for the very same reason that the equality of a NULL cannot be tested. If a column contains no value, you cannot say whether this matches "I don't know what".



```
   case upper(color)
     when 'Y' then 'Color'
     when 'N' then 'B&W'
➡    else '?'
   end as color,
   …
```

An unset value MIGHT be caught by an ELSE branch. There are cases, though, when relying on ELSE is impractical, for instance if you want to display something special when people are alive.



```
~~case died~~
~~  when 1920 then 'passed away'~~
~~  when 1921 then 'passed away'~~
~~  when 1922 then 'passed away'~~
~~  when 1923 then 'passed away'~~
~~  when 1924 then 'passed away'~~
~~  when 1925 then 'passed away'~~
~~  when 1926 then 'passed away'~~
~~  when 1927 then 'passed away'~~
~~  when 1928 then 'passed away'~~
~~  when 1929 then 'passed away'~~
```

You cannot have a WHEN branch for every possible year, past and future.

So there is a second form for CASE, in which CASE isn't followed by the name of the column to test, but each WHEN branch explicitly tests data.

```
case
  when died is null then
         'alive and kicking'
  else 'passed away'
end as status
```

This can also be used for conditions involving two columns from the same row, such as testing whether COL_A contains a value greater than the one in COL_B.

Some useful functions

A true DBMS product usually supplies dozens, sometimes hundreds, of functions. Let's have a brief look at the most commonly used ones.

Flickr:Sanath Kumar

Among numerical functions, trigonometric functions aren't the most used. round() and trunc(), though, are quite useful.

```
round(3.141592, 3)    3.142

trunc(3.141592, 3)    3.141
```

Related functions floor() and ceiling() (respectively the highest smaller and the lowest higher integer value) are also commonly used.

More string functions are used than numerical functions. They are often used for cleaning-up or standardizing data on entry.

```
upper(), lower()
substr('Citizen Kane', 5, 3)
trim(' Oops  ')          'Oops'
replace('Sheep', 'ee', 'i')
                          'Ship'
length()
```

One thing that is very special in SQL is date functions, for which every single DBMS have a different set. In business applications, there is a lot of computations on dates (for instance, computing when a bill is due).

Date Arithmetic

+ 1 month

≠

+ 30 days

Because of all the differences in number of days per month, leap years, and so forth, date arithmetic is everything but trivial. Hence functions, or special operators.

To give just one example, here is how you can add one month to a date (similar syntax for days, weeks, ... except for Oracle)

SQL Server	`dateadd(month, 1, date_col)`
IBM DB2	`date_col + 1 month`
PostgreSQL	`date_col + interval'1 month'`
MySQL	`date_add(date_col, interval 1 month)`

ORACLE	`add_months(date_col, 1)`
	`date_col + decimal_number`

SQLite	`date(date_col, '1 month')`

Character string **'28-DEC-1895'**

↓ format

DBMS date

As there is no other practical way to enter a date as a character string (other than as a number-of-whatever since a predefined origin) you need to convert, by specifying the format, a string into an internal date. Default formats vary by product, and can often be changed at the DBMS level.

Character string **'12/28/1895'**

↑ format

DBMS date

Conversely, you can format something that is internally stored as a date and turn it into a character string that has almost any format you want. If you know strftime() in C, it's the same idea (some products use formats that are the strftime() ones, other products have their own format) cast(as)

Finally, remember that when you compare values of different datatypes, some untold conversion will always take place, and will be silent if it succeeds. We'll see later that it sometimes has unpleasant consequences, and conversions should be made explicit, using a function such as cast(). Why do people compare different types? Modeling choices! Your student id is numerical, and might be stored in one table as a numerical value. At the same time, it has no numerical meaning and the designer of another table may have chosen to use a varchar(). As systems are slowly built over the years by different people, this type of inconsistency happens often.

Let's refine our queries ...

We have seen some very simple filtering conditions so far, it's time to change gear.

First of all, remember that numerical operations only work if some rules are respected. For instance, you can only divide z by x + y to obtain a new result if x + y is not zero.

$$x \quad y \quad z$$

$$\frac{z}{(x + y)}$$

No duplicates

Identifier

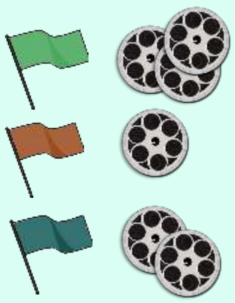
Same story with relational operations. Some rules must be respected if you want to obtain valid results when you apply new operations to result sets (they must be mathematical sets).

```
select country from movies
```

The problem is that if we run a query such as this one, we'll see many, many identical rows. In other words, we may be obtening a table, but it's not a relation because many rows cannot be distinguished.

The result of the previous query is in fact completely uninteresting. Whenever we are only interested in countries in table movies, without paying attention to anything else, it can only be for one of two reasons:

- We want to see a list of countries for which we have films,
- or we want for instance see which countries appear most often



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

distinct

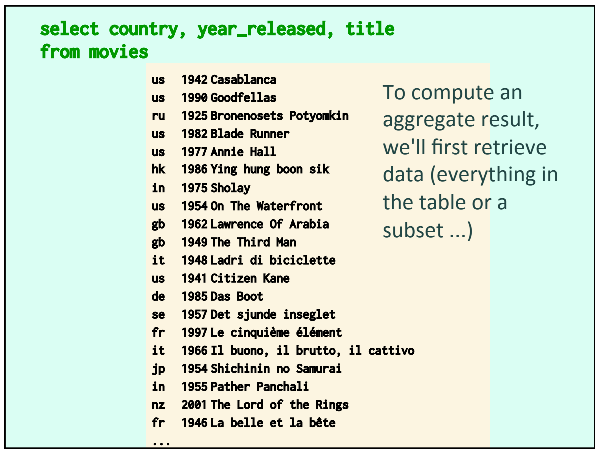
```
select distinct country
from movies
```

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.

When we are interested in what we might call country-wide characteristics, we use

Aggregate functions

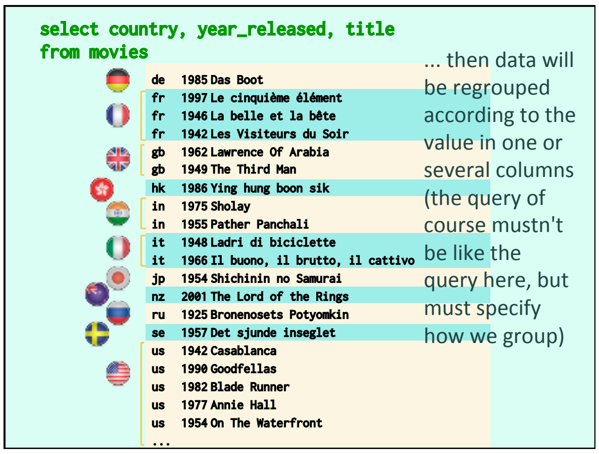
As the name says, aggregate function will aggregate all rows that share a feature (such as being films from the same country) and return a characteristic of each group of aggregated rows. It will be clearer with an example.



```
select country, year_released, title
from movies
```

```
us   1942 Casablanca
us   1990 Goodfellas
ru   1925 Bronenosets Potyomkin
us   1982 Blade Runner
us   1977 Annie Hall
hk   1986 Ying hung boon sik
in   1975 Sholay
us   1954 On The Waterfront
gb   1962 Lawrence Of Arabia
gb   1949 The Third Man
it   1948 Ladri di biciclette
us   1941 Citizen Kane
de   1985 Das Boot
se   1957 Det sjunde inseglet
fr   1997 Le cinquième élément
it   1966 Il buono, il brutto, il cattivo
jp   1954 Shichinin no Samurai
in   1955 Pather Panchali
nz   2001 The Lord of the Rings
fr   1946 La belle et la bête
...
```

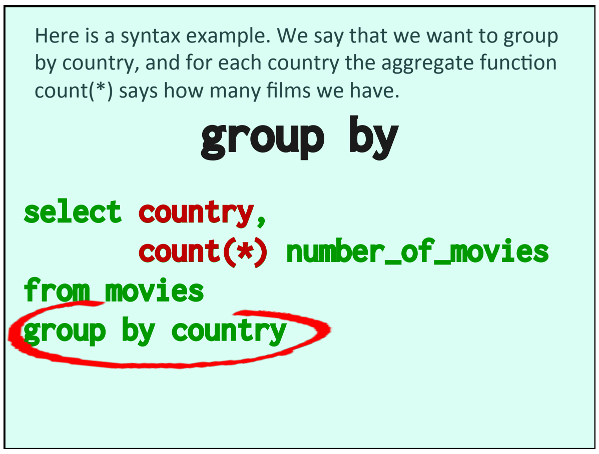
To compute an aggregate result, we'll first retrieve data (everything in the table or a subset ...)



```
select country, year_released, title
from movies
```

```
de   1985 Das Boot
fr   1997 Le cinquième élément
fr   1946 La belle et la bête
fr   1942 Les Visiteurs du Soir
gb   1962 Lawrence Of Arabia
gb   1949 The Third Man
hk   1986 Ying hung boon sik
in   1975 Sholay
in   1955 Pather Panchali
it   1948 Ladri di biciclette
it   1966 Il buono, il brutto, il cattivo
jp   1954 Shichinin no Samurai
nz   2001 The Lord of the Rings
ru   1925 Bronenosets Potyomkin
se   1957 Det sjunde inseglet
us   1942 Casablanca
us   1990 Goodfellas
us   1982 Blade Runner
us   1977 Annie Hall
us   1954 On The Waterfront
...
```

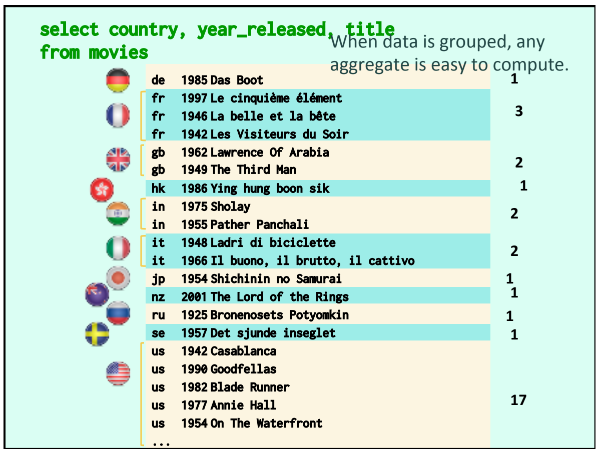
... then data will be regrouped according to the value in one or several columns (the query of course mustn't be like the query here, but must specify how we group)



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 films we have.

group by

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



```
select country, year_released, title
from movies
```

When data is grouped, any aggregate is easy to compute.

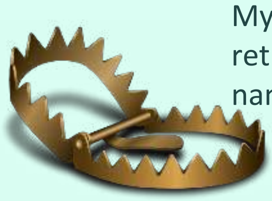
```
de   1985 Das Boot                            1
fr   1997 Le cinquième élément
fr   1946 La belle et la bête                 3
fr   1942 Les Visiteurs du Soir
gb   1962 Lawrence Of Arabia                  2
gb   1949 The Third Man
hk   1986 Ying hung boon sik                  1
in   1975 Sholay                              2
in   1955 Pather Panchali
it   1948 Ladri di biciclette                 2
it   1966 Il buono, il brutto, il cattivo
jp   1954 Shichinin no Samurai                1
nz   2001 The Lord of the Rings               1
ru   1925 Bronenosets Potyomkin               1
se   1957 Det sjunde inseglet                 1
us   1942 Casablanca
us   1990 Goodfellas
us   1982 Blade Runner                        17
us   1977 Annie Hall
us   1954 On The Waterfront
...
```

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.

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

Most products generate an error for this query

```
select continent,
       count(*) num_of_countries
from countries
```

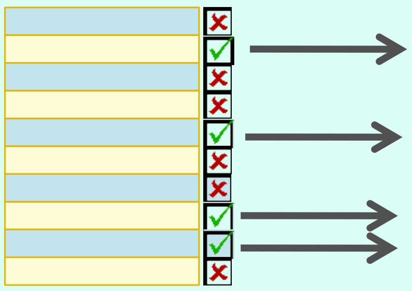


MySQL and SQLite return no error and the name of one continent (ANY of them)

However, it is perfectly valid to have nothing other than one (or several) aggregate functions after SELECT, in which case no GROUP BY is required. What is computed is for the whole of the table.

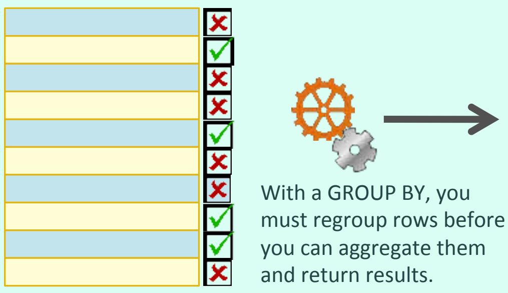
```
select count(*) number_of_movies
from movies
```

where



Beware of some performance implication. When you apply a simple WHERE filter, you can start returning rows as soon as you have found a match.

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.

count(*) / count(col)

min(col) stddev()

max(col)

avg(col)

sum(col)

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.

Earliest release year by country?

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

Such a query answers the question. Note that in the demo database years are simple numerical values, but generally speaking min() applied to a date logically returns the earliest one. The result will be a relation: no duplicates, and the key that identifies each row will be the country code (generally speaking, what follows GROUP BY).

Therefore we can validly apply another relational operation such as the "select" operation (row filtering) and only return countries for which the earliest movie was released before 1940.

```
select *
from (
select country,
       min(year_released) oldest_movie
from movies
group by country
) earliest_movies_per_country
where oldest_movie < 1940
```

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)

The following query is perfectly valid in SQL. What you are doing is aggregating films for all countries, then discarding everything that isn't American:

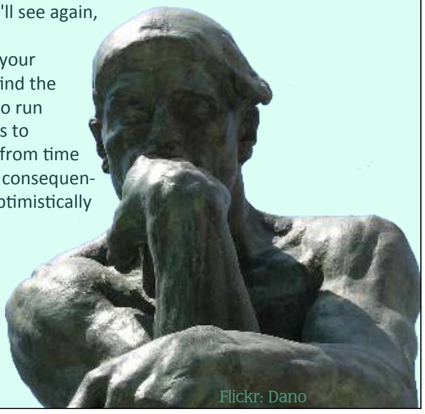
```
select country,
       min(year_released) oldest_movie
from movies
group by country
having country = 'us'
```

The efficient way to proceed is of course to select American films first, and only aggregate them:

```
where country = 'us'
```

SQL Server will do the right thing behind your back. Oracle will assume that you have some obscure reason for writing your query that way and will do as told. It can hurt.

All database management systems have a highly important component that we'll see again, called the "query optimizer". It takes your query, and tries to find the most efficient way to run it. Sometimes it tries to outsmart you, with from time to time unintended consequences, sometimes it optimistically assumes that you know what you are doing. Optimizers don't all behave the same.



If you want no trouble whichever DBMS you are using, ALWAYS apply HAVING to the <u>result</u> of an aggregate, something that you cannot know beforehand.

having

Result of aggregate

Nulls?

known + unknown = unknown

When you apply a function or operators to a null, with very few exceptions the result is null because the result of a transformation applied to something unknown is an unknown quantity. What happens with aggregates?

Aggregate functions

ignore

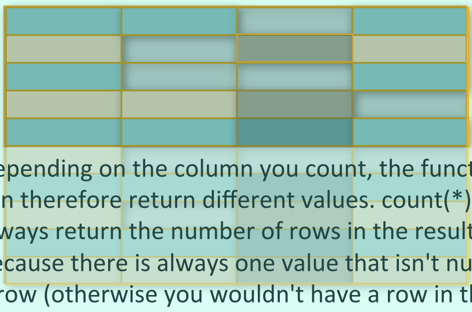
Nulls

FLickr: Linda Åslund

```
select max(died) most_recent_death
from people
where died is not null
```

In this query, the WHERE condition changes nothing to the result (perhaps it makes more obvious that we are dealing with dead people only, but for the SQL engine it's implicit)

count(*) count(col)



Depending on the column you count, the function can therefore return different values. count(*) will always return the number of rows in the result set, because there is always one value that isn't null in a row (otherwise you wouldn't have a row in the first place)

```
select count(*) people_count,
       count(born) birth_year_count,
       count(died) death_year_count
from people
```

Counting a mandatory column such as BORN will return the same value as COUNT(*). The third count, though, will only return the number of dead people in the table.

```
select count(colname)
select count(distinct colname)
```

In some cases, you only want to count distinct values. For instance, you may want to count how many different surnames start with a Q instead of how many people have a surname that starts with a Q.

These two queries are equivalent

```
select country,
       count(distinct year_released)
number_of_years
from movies
group by country

select country,
       count(*) number_of_years
from (select distinct country,
             year_released
      from movies) t
group by country
```

Here we'll only get one row per country and year

A good illustration of answering a question that isn't so easy is the following one.

How many people **?**
are both
actors and directors?

credits

First thing is to correctly identify where we'll find the information. "Actor" and "Director" is in CREDITS. It only contains PEOPLEID values but they are OK to count. We don't need names.

~~movieid~~	peopleid	credited_as	
~~8~~	37	D	
~~8~~	38	A	⬅
~~8~~	39	A	
~~8~~	40	A	
~~10~~	11	A	⬅
~~10~~	12	A	
~~10~~	15	D	
~~10~~	16	A	
~~10~~	17	A	
~~12~~	11	A	⬅
~~12~~	11	D	
~~12~~	12	A	
~~135~~	378	D	
~~136~~	433	A	
~~136~~	434	A	
~~136~~	435	A	
~~115~~	38	A	⬅
~~115~~	359	D	
~~115~~	360	A	
~~...~~			

```
select peopleid,
       credited_as
from credits
```

There is no restriction such as "that have played in a film that they have directed", so the movieid is irrelevant. But if we remove the movieid, we have tons of duplicates. Not a relation!

People who appear twice are the ones we want.

peopleid	credited_as
11	A
11	D
12	A
15	D
16	A
17	A
37	D
38	A
39	A
40	A
359	D
360	A
361	A
378	D
379	A
380	A
442	A
442	D
443	A
...	

```
select distinct
       peopleid,
       credited_as
from credits
where credited_as
   in ('A', 'D')
```

DISTINCT will remove duplicates and provide a true relation. I specify the values for CREDITED_AS because there are no other values now but you can't predict the future (someday there may be producers or directors of photography).

```
select count(*) number_of_acting_directors
from (
select peopleid, count(*) as number_of_roles
from (select distinct
             peopleid,
             credited_as
      from credits
      where credited_as
         in ('A', 'D')) all_actors_and_directors
group by peopleid
having count(*) = 2) acting_directors
```

The HAVING selects only people who appear twice ... and we just have to count them. Mission accomplished.