CS213
Principles of Database Systems(H)

Chapter 3
Retrieving Data from One Table

Shiqi YU 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

3.1 Select

Shiqi Yu 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

`select * from tablename`

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

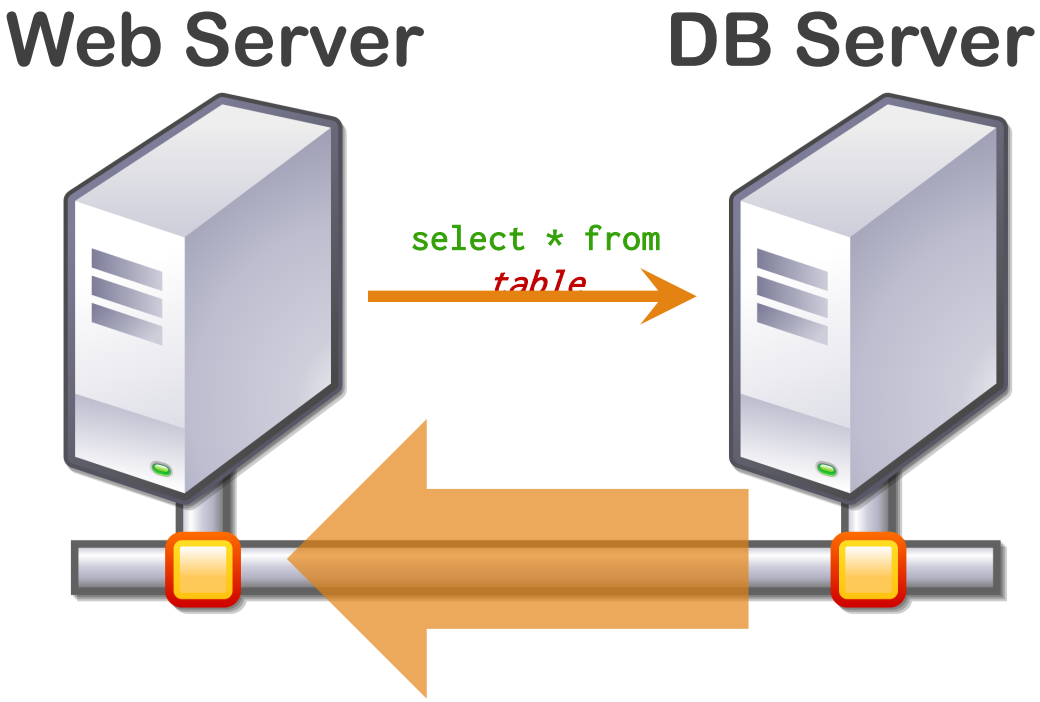
* is short-hand for "all columns" and is frequently used in interactive tools (especially when you don't remember column names ...)

You should not use it, though, in programs.

```
select * from table
```

≈

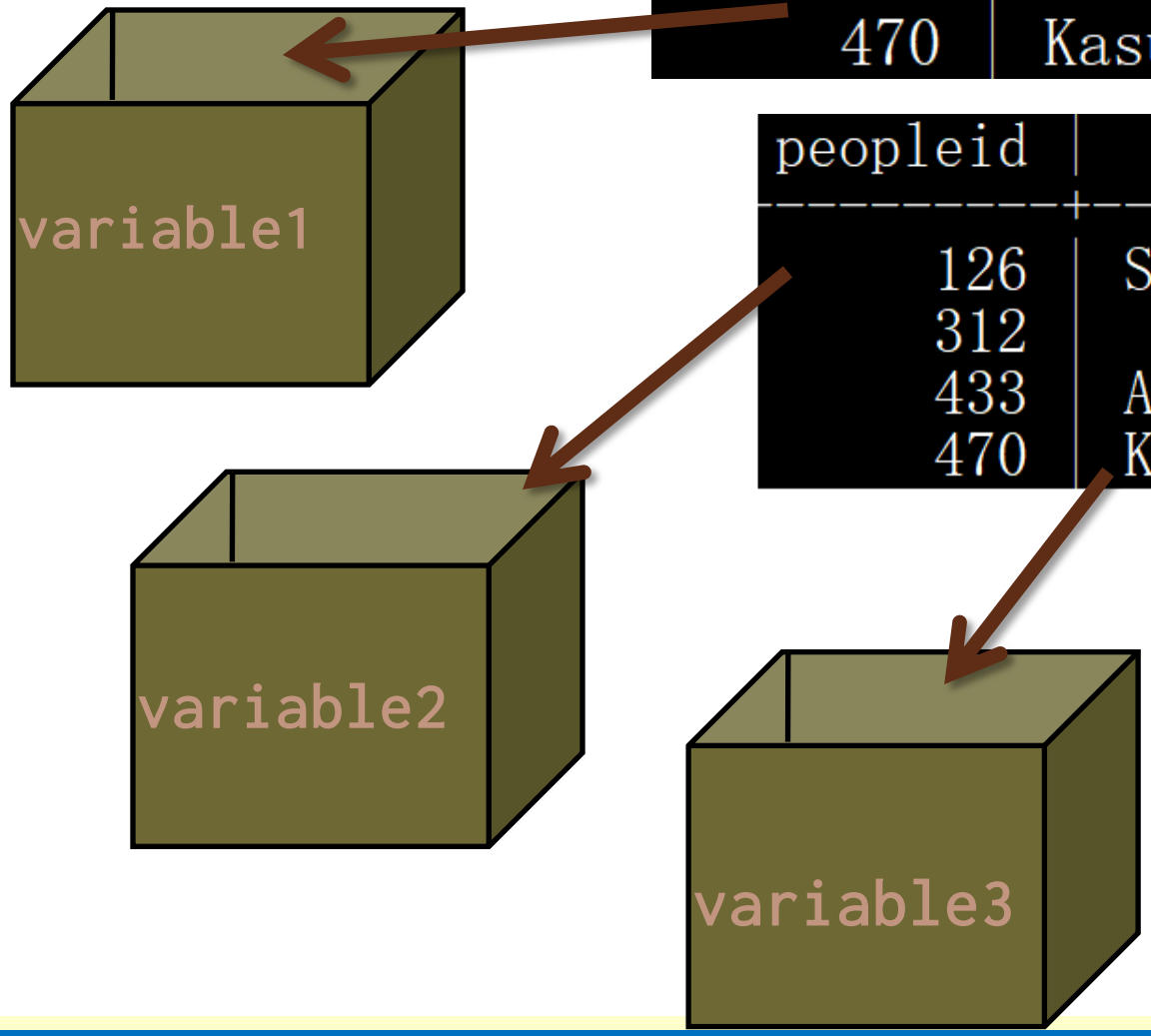
```
print table
```



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

Row Data

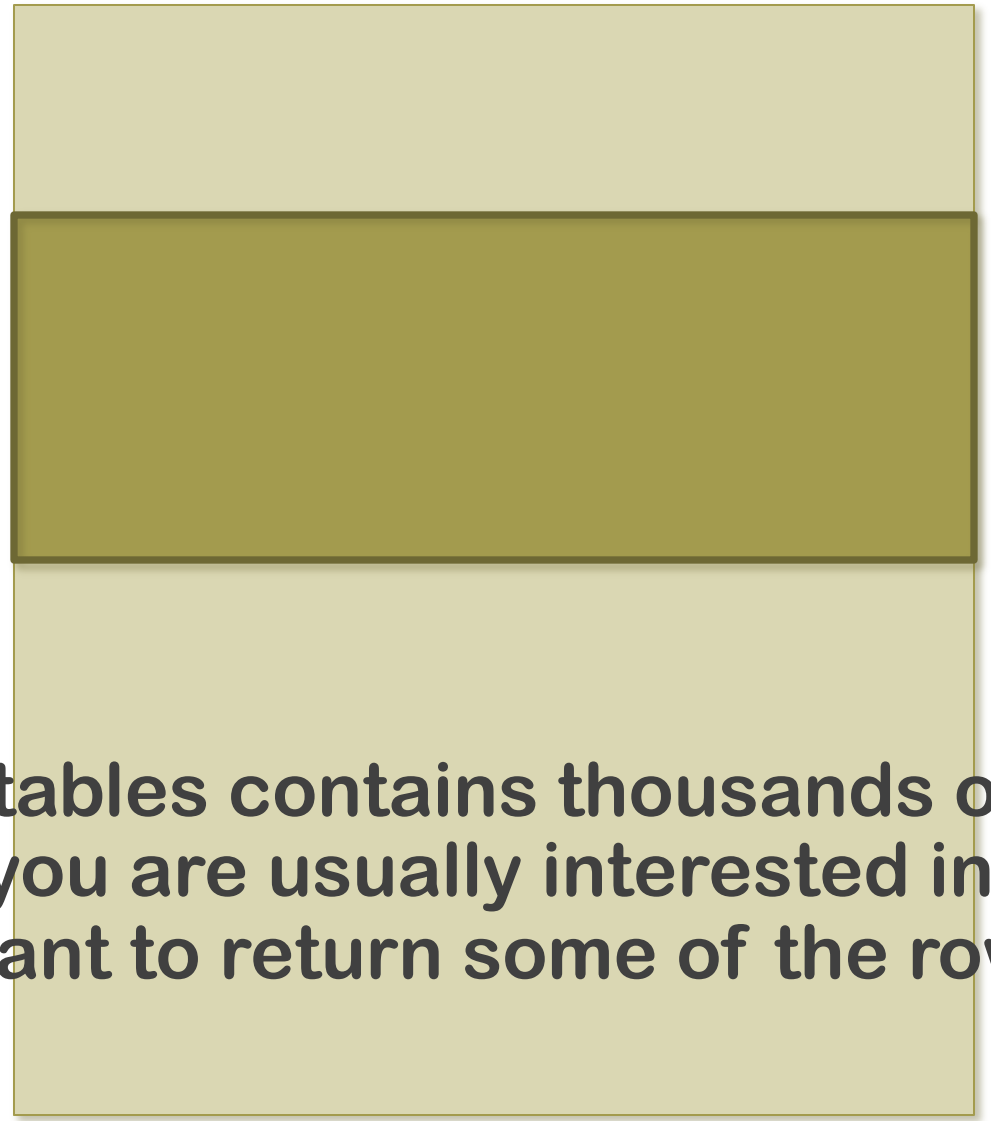
peopleid	name	born
126	Seo-hyun Ahn	2004
312	Anandhi	1993
433	Atsushi Arai	1994
470	Kasumi Arimura	1993



peopleid	first_name	surname	born
126	Seo-hyun	Ahn	2004
312		Anandhi	1993
433	Atsushi	Arai	1994
470	Kasumi	Arimura	1993

In a program you want to retrieve every column in a variable. Don't forget ALTER: you can add columns to a table. One day it may break your program, or make it fetch unneeded data and use bandwidth for nothing. In a program, always name columns.

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)

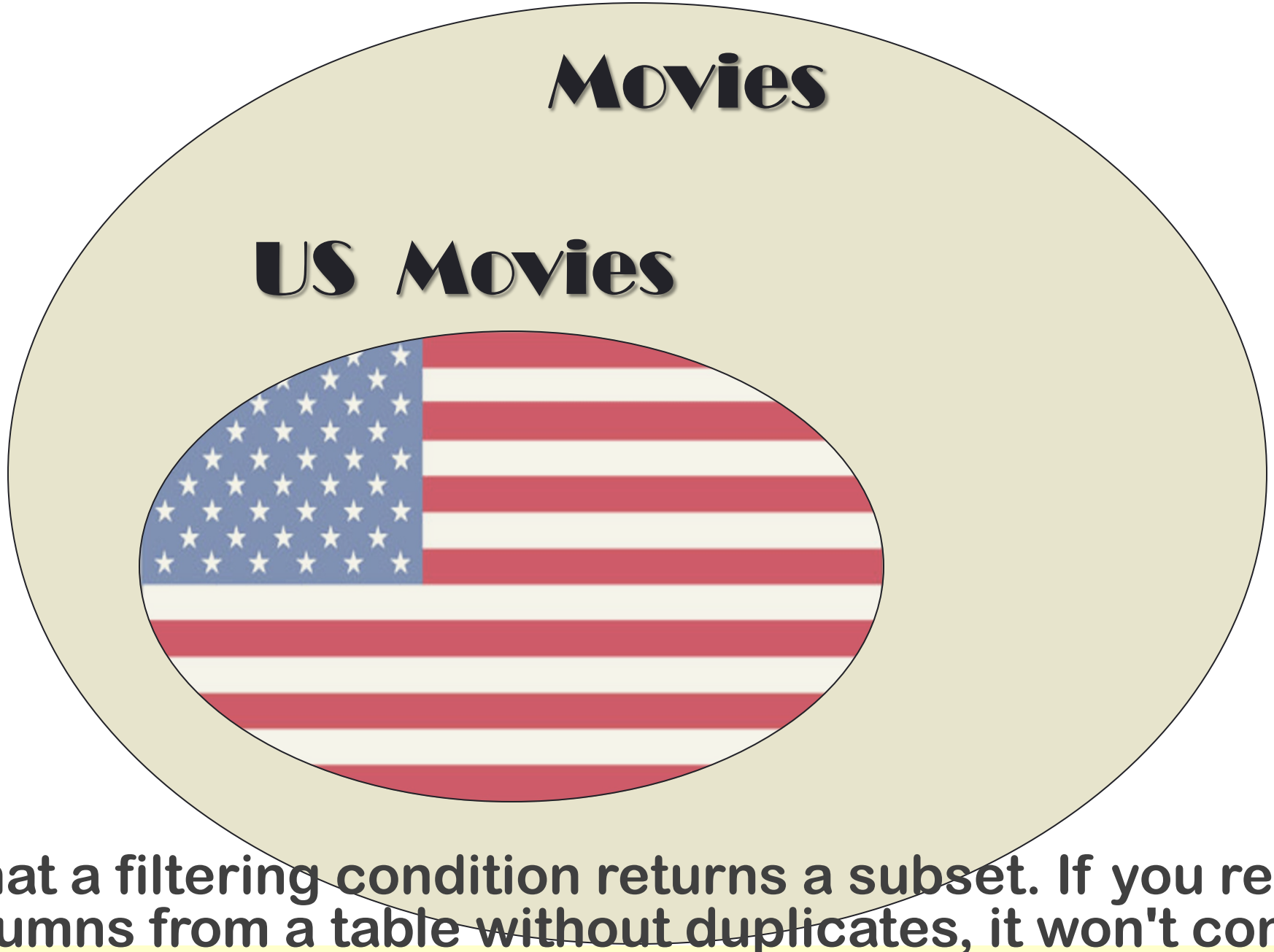
```
filmdb=# select * from people where first_name = surname;
 peopleid | first_name | surname | born | died | gender
----------+------------+---------+------+------+--------
     4270 | Evans      | Evans   | 1936 |      | F
     5745 | Han        | Han     | 1982 |      | M
    10681 | Ni         | Ni      | 1988 |      | F
    15306 | Wan        | Wan     | 1981 |      | F
    16015 | Yang       | Yang    | 1991 |      | M
```

Column doesn't exist!

```
where surname = Han
where surname = 'Han'
```

```
filmdb=# select * from people where surname=Han;
ERROR:  column "han" does not exist
LINE 1: select * from people where surname=Han;
                                           ^
```

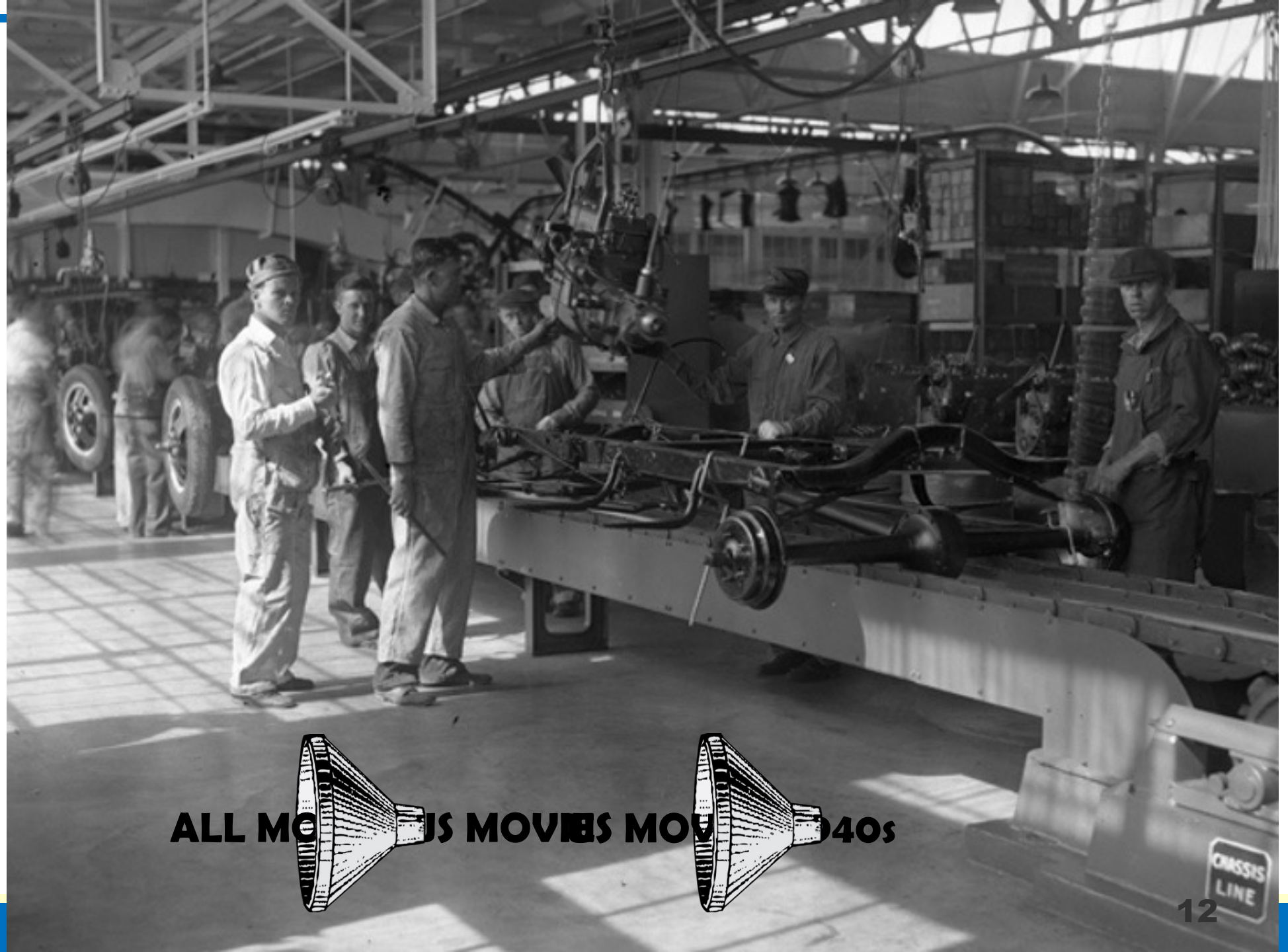
Beware that string constants must be quoted between single-quotes. If they aren't quoted, they will be interpreted as column names. Same thing with Oracle if they are double-quoted.



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.


CHASSIS
LINE

```
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

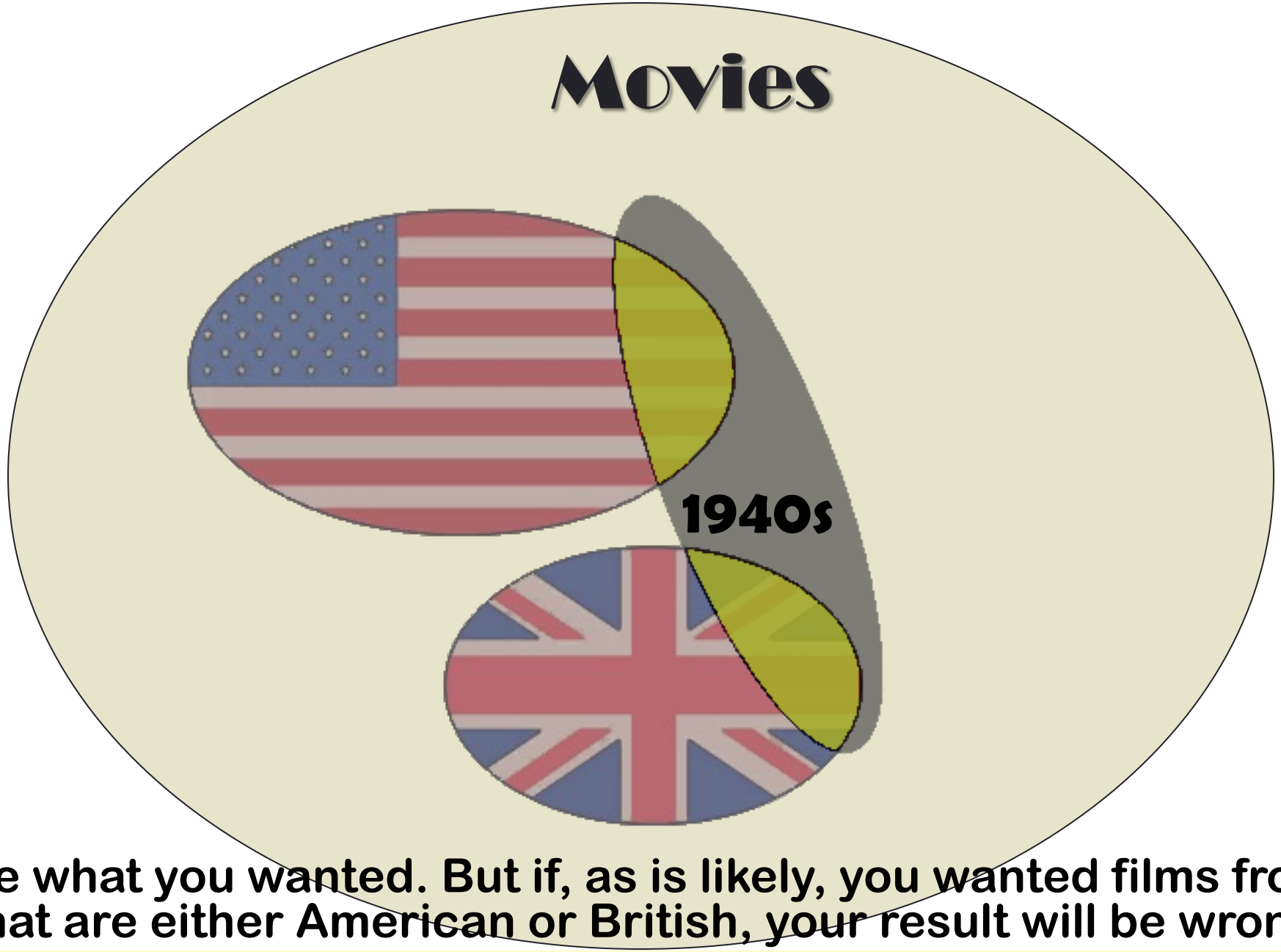
You can also link conditions with OR and negate them with NOT. Beware of boolean logic: in a program, an IF or WHILE rarely contains more than 2 or 3 conditions. An SQL query can contain dozens, and boolean logic can become quite complicated.

and > or

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

```
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

Chinese movies from the 1940s

plus

American movies from the 1950s

```
select * from movies
where (country = 'cn'
  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.

The parentheses make the statement easier to understand.

=

<> 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.

```
2 < 10
```

~~'2' < '10'~~

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

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

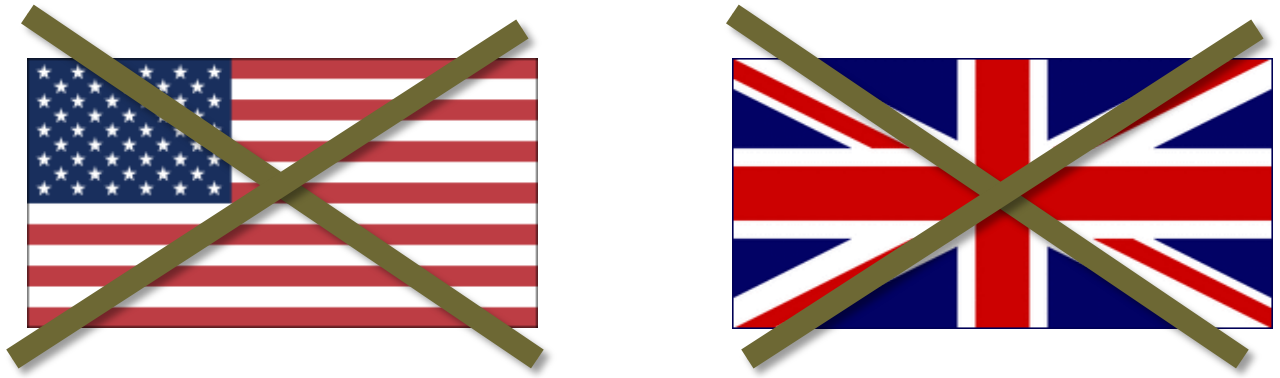
As strings!

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

Another way to handle case is to force it, but performancewise 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.

3.2 Date

Shiqi Yu 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

DD/MM/YYYY

MM/DD/YYYY

YYYY/MM/DD

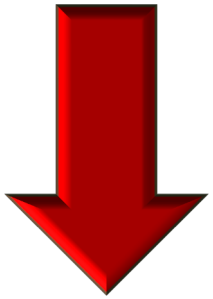
Beware also of date formats, and of conflicting European/American formats which can for some dates be ambiguous. Common problem in multinational companies.

Convert EXPLICITLY!

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.

```
select * from forum_posts where post_date>= '2018-03-12';
select * from forum_posts where post_date>=date('2018-03-12');
select * from forum_posts where post_date>=date('12 March, 2018');
```

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'

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)

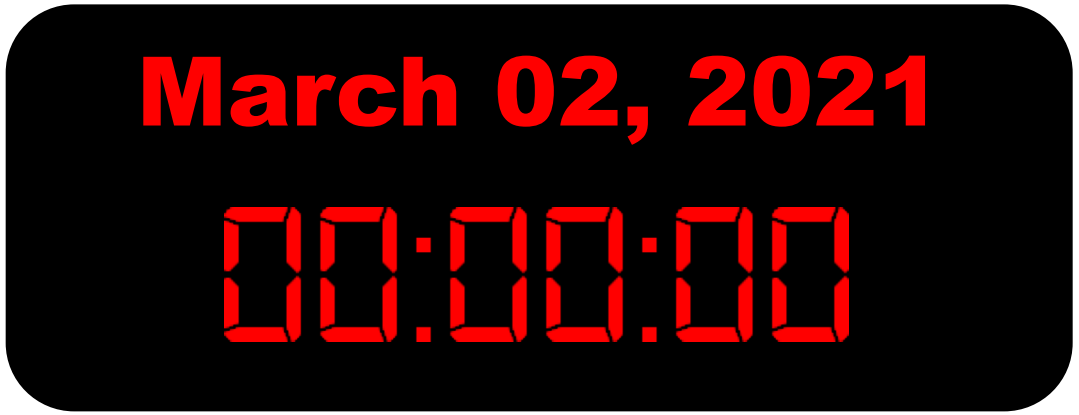
Another frequent mistake is with **datetime** values. If you compare them to a **date** (without any time component) the SQL engine will not understand that the date part of the datetime should be equal to that date.

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

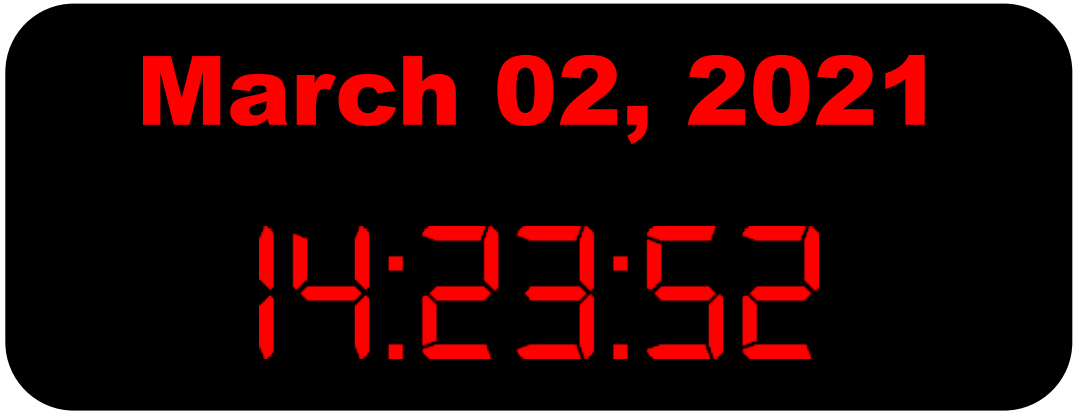
Flickr:Yoppy & Rudolf Schuba

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

Date: March 02, 2021



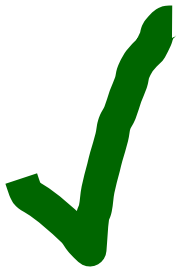
Datetime: March 02, 2021 14:23:52

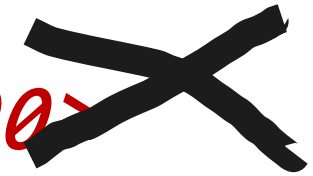


Everything that won't have happened at that precise second won't satisfy the condition and won't be returned.

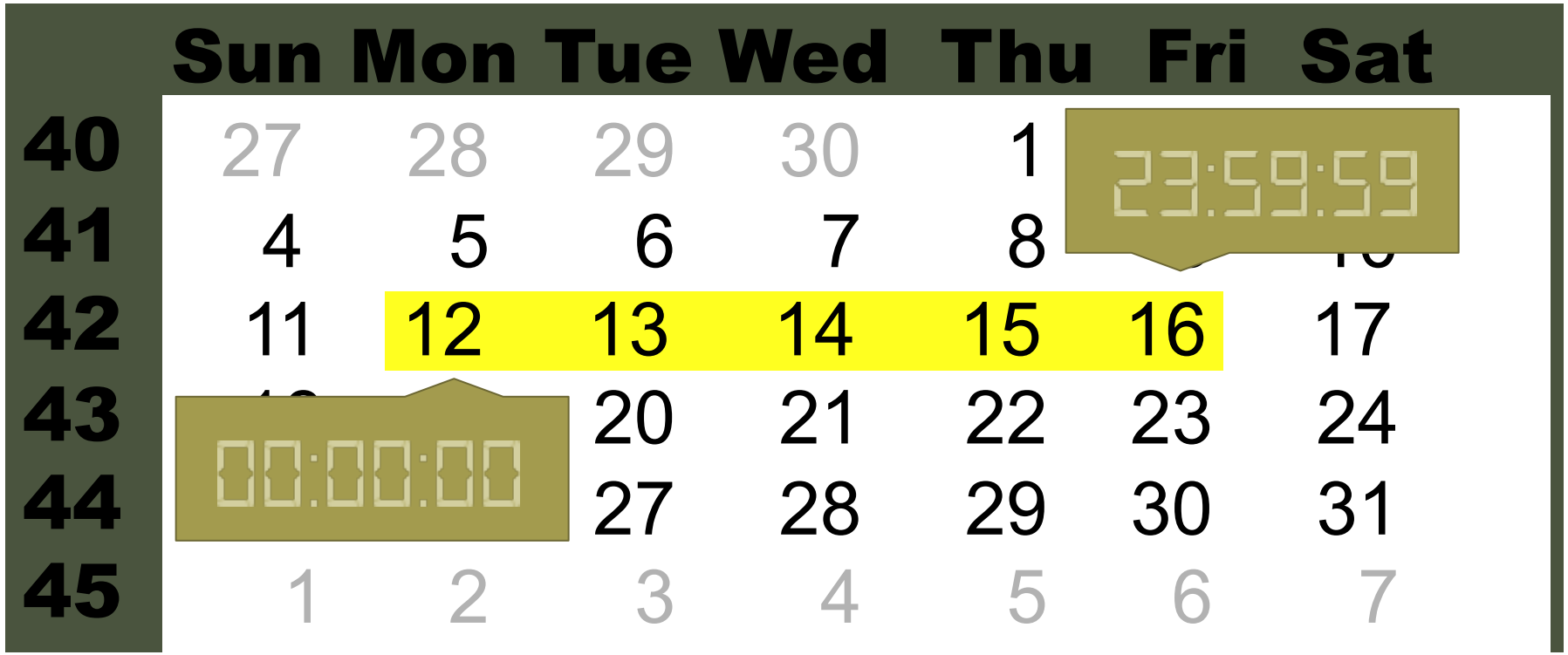
Datetime type

```
where issued >= <March 12>
                <March 12 00:00:00>
  and issued <= <March 16>
                <March 16 00:00:00>
```





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 March 16, which probably wasn't the intent.



A correct condition would select everything from March 12th at 00:00:00 to March 16th at 23:59:59 included, or March 17th 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**.

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.


```
dateadd(month, 1, date_col)
```


```
dateadd(month, 1, date_col)
```


```
date_col + 1 month
```


```
dateadd(month, 1, date_col)
```


```
date_col + 1 month
```


```
date_col + interval'1 month'
```

```
filmdb=# select date('2020-01-30')+interval'1 month';
      ?column?
---------------------
 2020-02-29 00:00:00
(1 row)
```


```
dateadd(month, 1, date_col)
```


```
date_col + 1 month
```


```
date_col + interval'1 month'
```


```
date_add(date_col, interval 1 month)
```

Microsoft 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
```

days


```
dateadd(month, 1, date_col)
```


```
date_col + 1 month
```


```
date_col + interval'1 month'
```


```
date_add(date_col, interval 1 month)
```


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


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

3.3 NULL

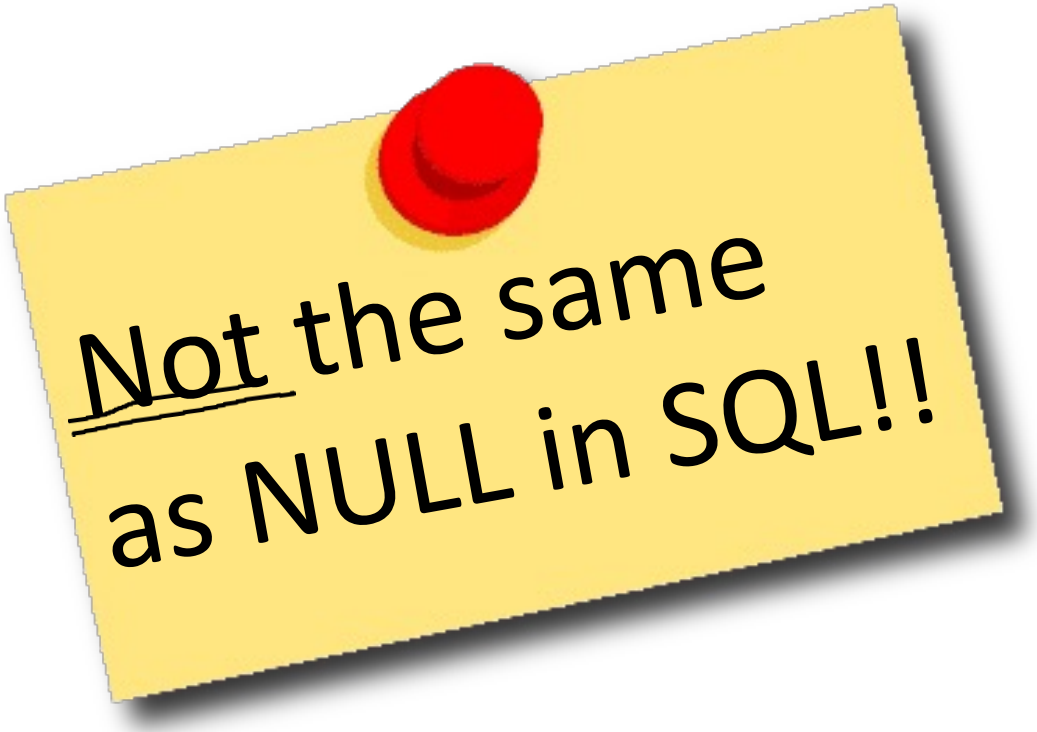
Shiqi Yu 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

In a language such as Java, you can compare a reference to *null*, because *null* is defined as the '0' address. In C, you can also compare a pointer to NULL (*pointer* is C-speak for reference). Not in SQL, where NULL denotes that **a value is missing.**

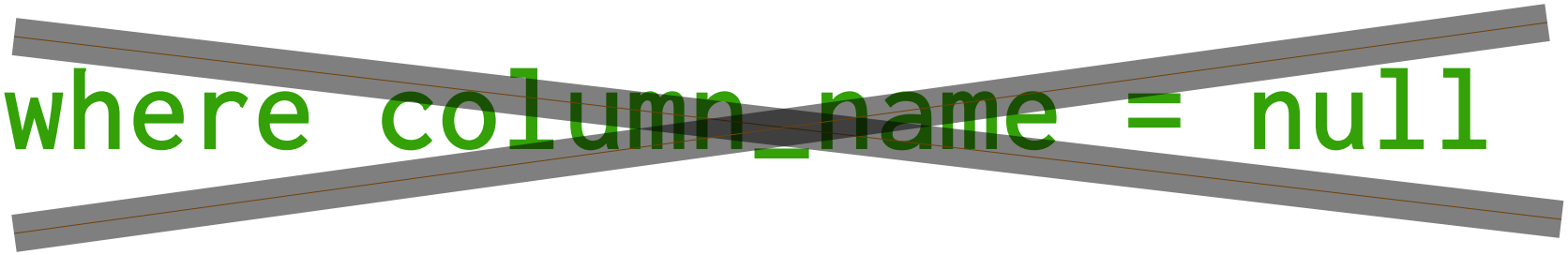
```
if (ptr == NULL) {
    ...
```



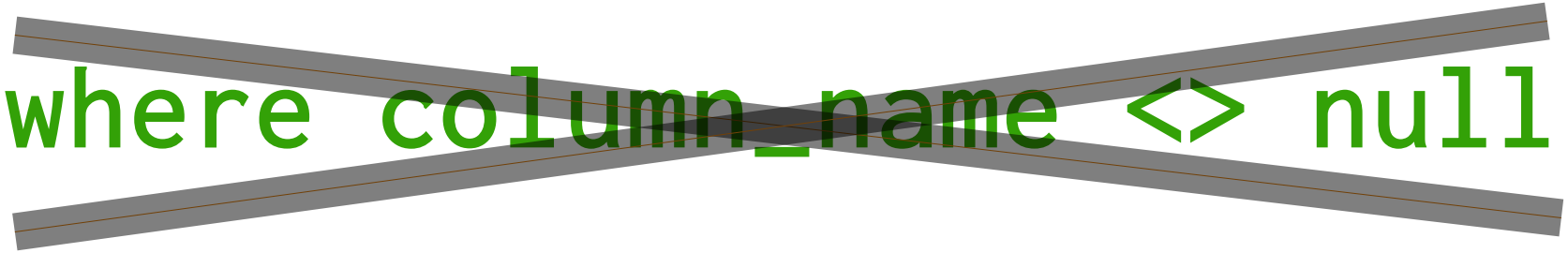
NULL in SQL
is NOT a value ...

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
```

Who are the people in the database who are not alive?

```
select * from people
   where died is not null
```

3.4 Some Functions

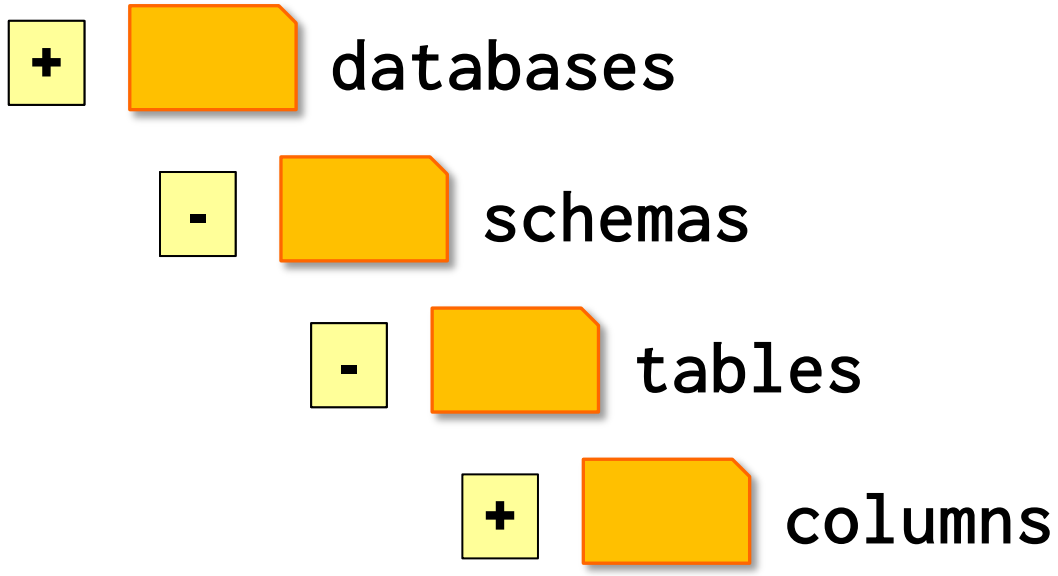
Shiqi Yu 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

```
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
```

compute derive

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.

```
select title
       || ' was released in '
       || cast(year_released as varchar)
movie_release
from movies
where country = 'us'
```

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!

people

born

died

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

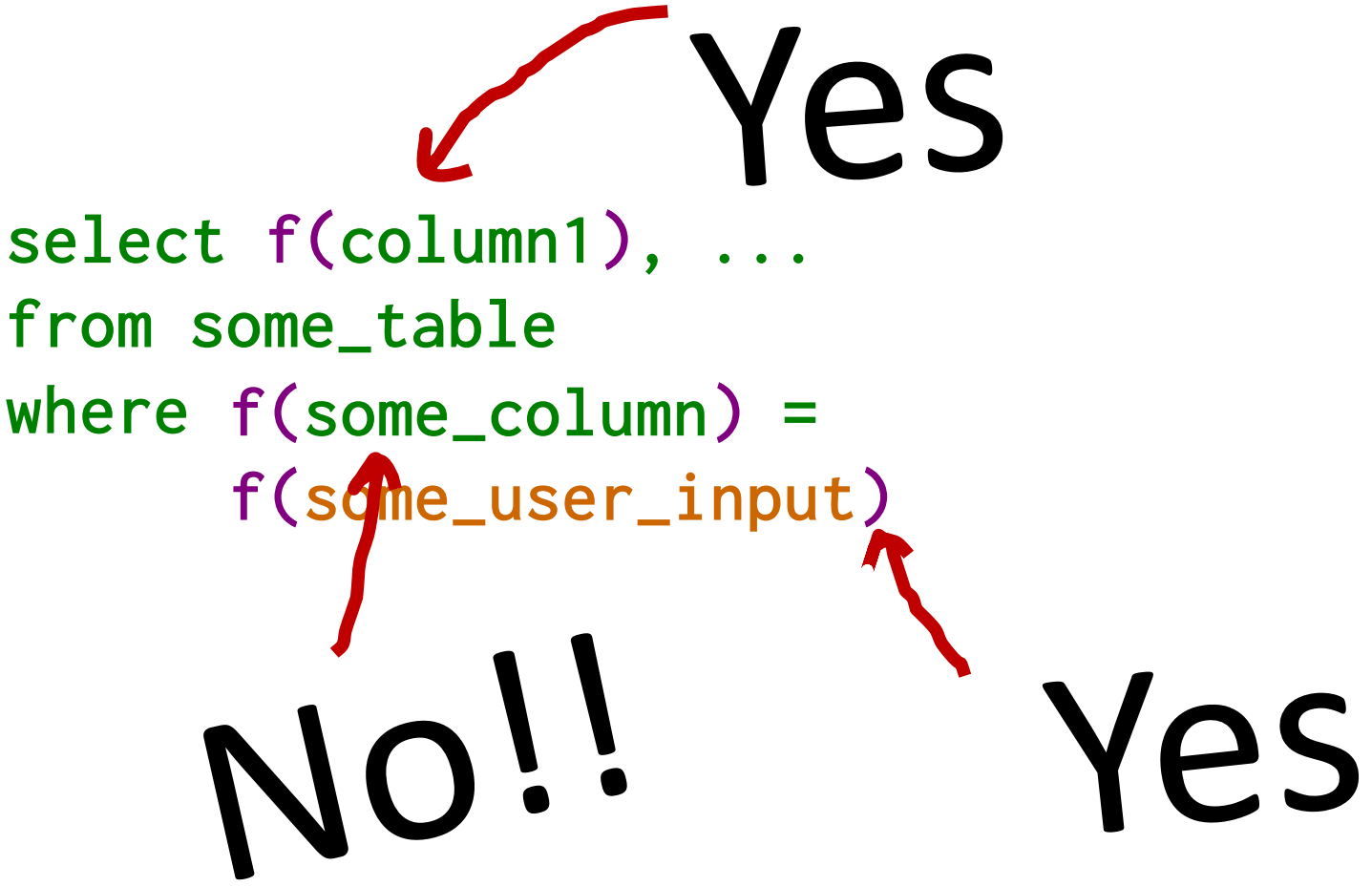
```
died  is null
```

Age

```
<this year> -  born
```

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

```
select peopleid, surname,
  date_part('year', now())-born as age
from people
where died is null
```



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.

Some useful functions

Flickr:Sanath Kumar

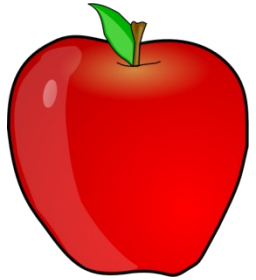
Among numerical functions, trigonometric functions aren't the most used. round() and trunc(), though, are quite useful.

```
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.

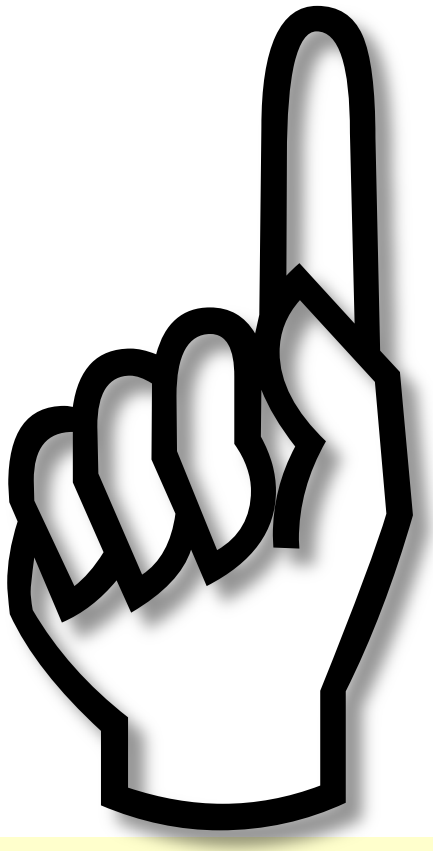
```
upper(), lower()

substr('Citizen Kane', 5, 3) 'zen'
        ↑
        1

trim(' Oops  ')          'Oops'

replace('Sheep', 'ee', 'i') 'Ship'
```

cast( as ) 

Run the following statements to understand cast()

```
select cast(born as char)||'abc' from people;
select cast(born as char(2))||'abc' from people;
select cast(born as char(10))||'abc' from people;
select cast(born as varchar)||'abc' from people;
select cast(born as varchar(2))||'abc' from people;
```

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.



$$x \quad y \quad z$$

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

3.5 Case

Shiqi Yu 于仕琪

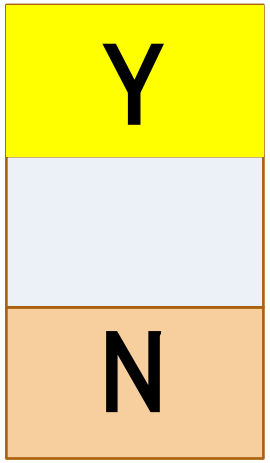
yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

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

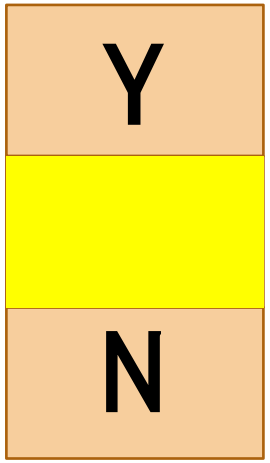


Color

```
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.

color

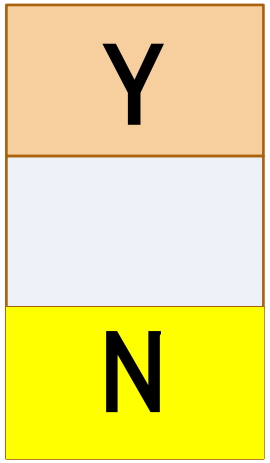


Color

?

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

color



Color

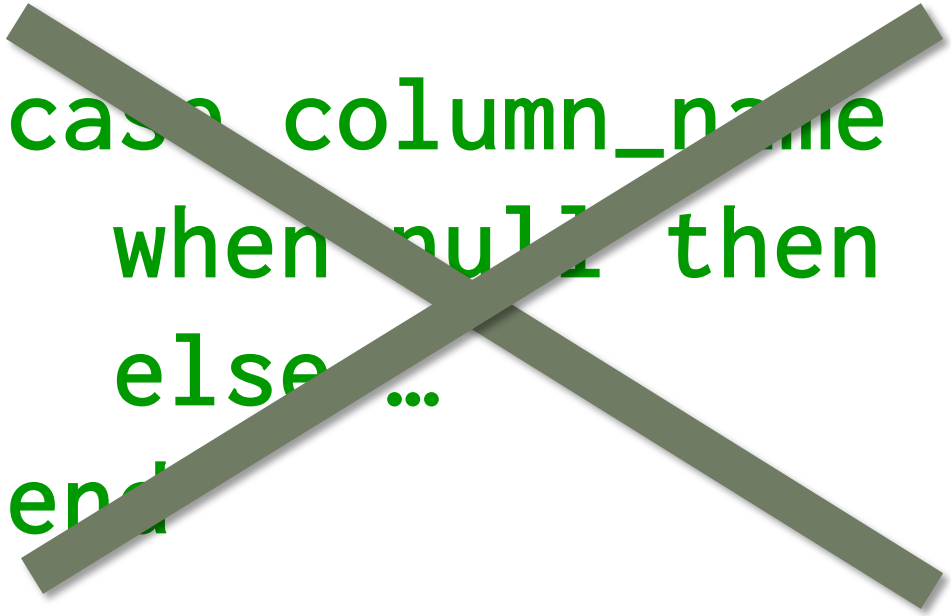
?

B&W

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

What if there is no "else" ?

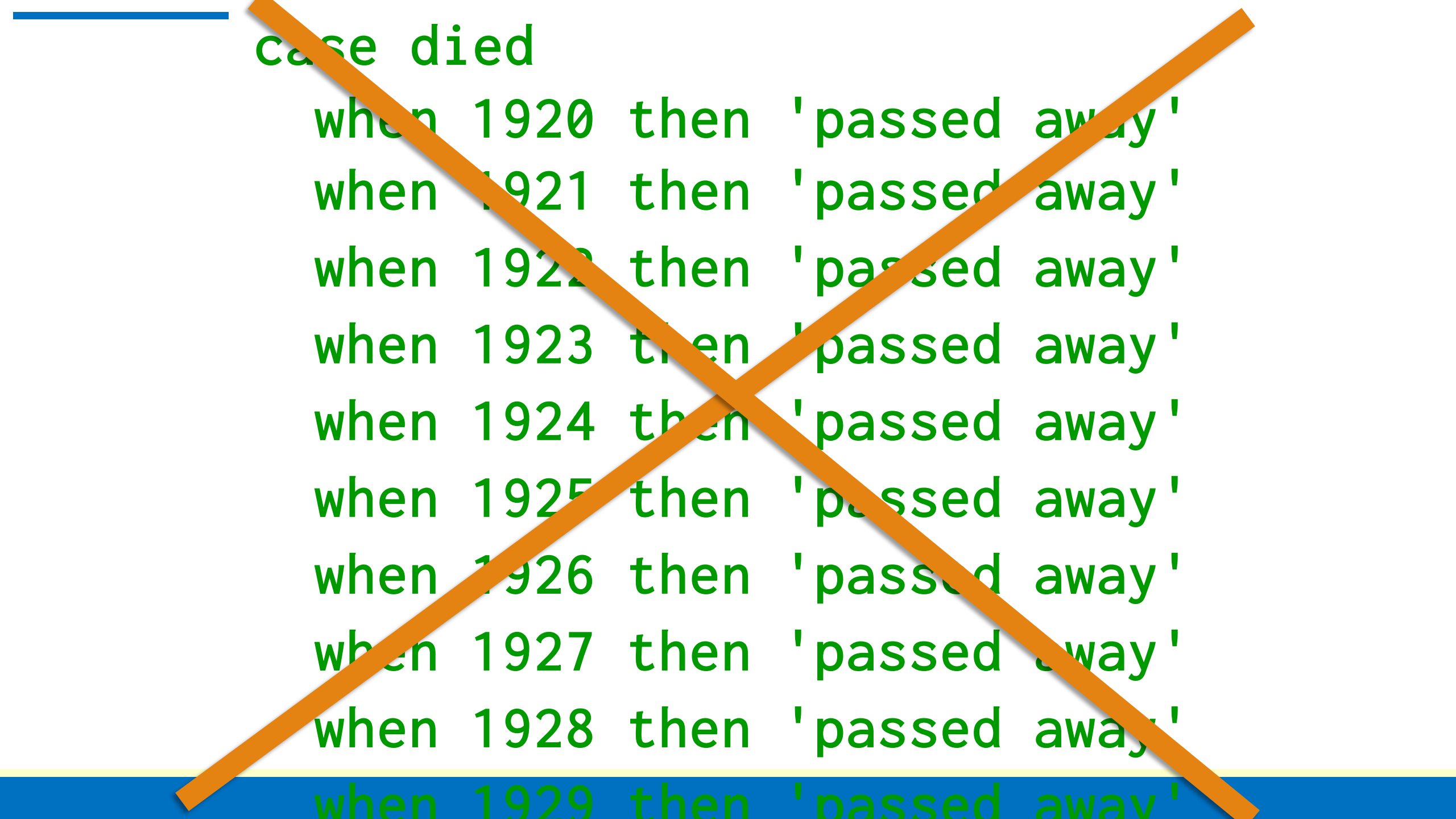
Remember NULLs?



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'

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.

surname	status
	alive and kicking
	passed away
50 Cent	alive and kicking
Aaliyah	passed away
Aamani	alive and kicking
Aames	alive and kicking
Aaron	alive and kicking
Aaron	alive and kicking
Abashidze	passed away
Abatantuono	alive and kicking
Abbad	alive and kicking
Abbas	passed away
Abbass	alive and kicking
Abbott	passed away
Abbott	alive and kicking
Abbott	passed away
Abbott	passed away
Abdalla	alive and kicking
Abdi	alive and kicking
Abdul-Jabbar	alive and kicking
Abdulov	passed away
Abe	alive and kicking
Abe	alive and kicking
Abe	passed away
Abe	alive and kicking
Abel	passed away
Abel	alive and kicking
Abel	alive and kicking
Abelanski	alive and kicking

```
select surname,
  case
    when died is null then
              'alive and kicking'
    else 'passed away'
  end as status
from people
```