CS307
Database Principles

Chapter 4

Most contents are from Stéphane Faroult's slides

4.1 Distinct

Most contents are from Stéphane Faroult's slides

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).

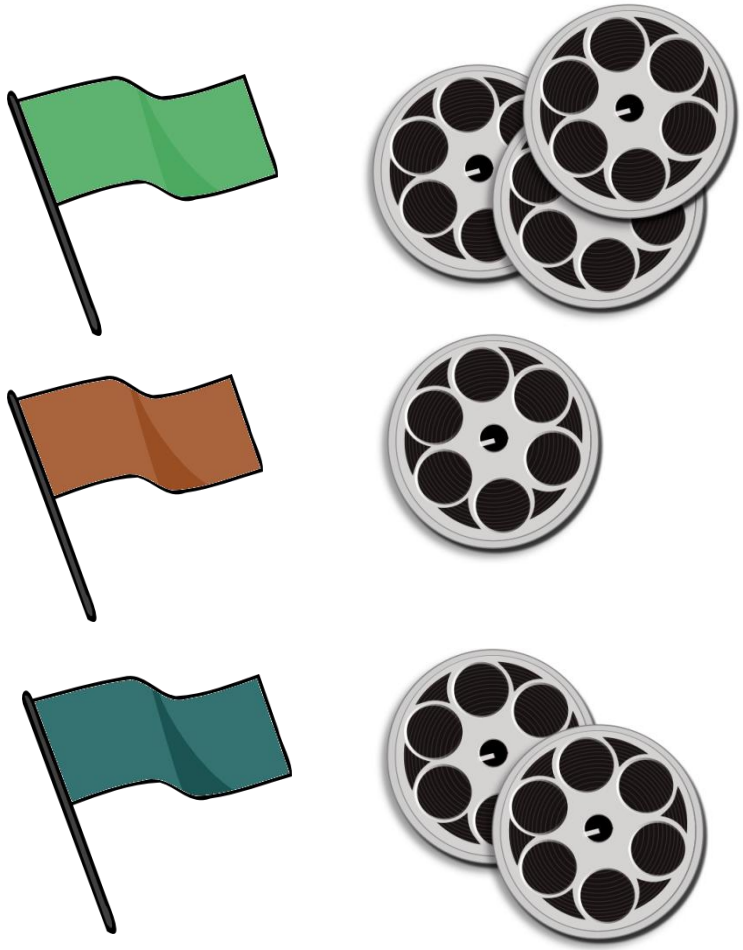
```
country
--------
ma
ar
hk
us
hk
us
mx
us
gb
us
us
ir
us
us
us
us
us
us
us
us
us
us
us
us
sp
se
```

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

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 obtaining 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:

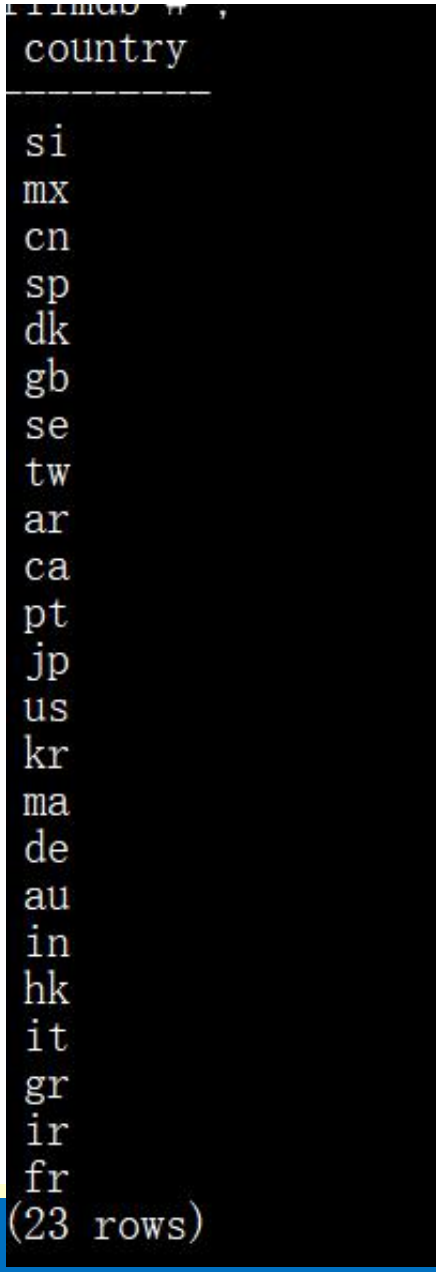
- We want to see a list of countries for which we have movies,
- or we want for instance see which countries appear most often



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

```
select distinct country from
movies
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

4.2 Aggregate Functions

Most contents are from Stéphane Faroult's slides

When we are interested in what we might call countrywide characteristics, such as how many movies released, we use

Aggregate functions

As the name says, aggregate function will aggregate all rows that share a feature (such as being movies 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
cn	1965	Da Nao Tian Gong
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
cn	1965	Da Nao Tian Gong
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 movies we have.

group by

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

One row for each group

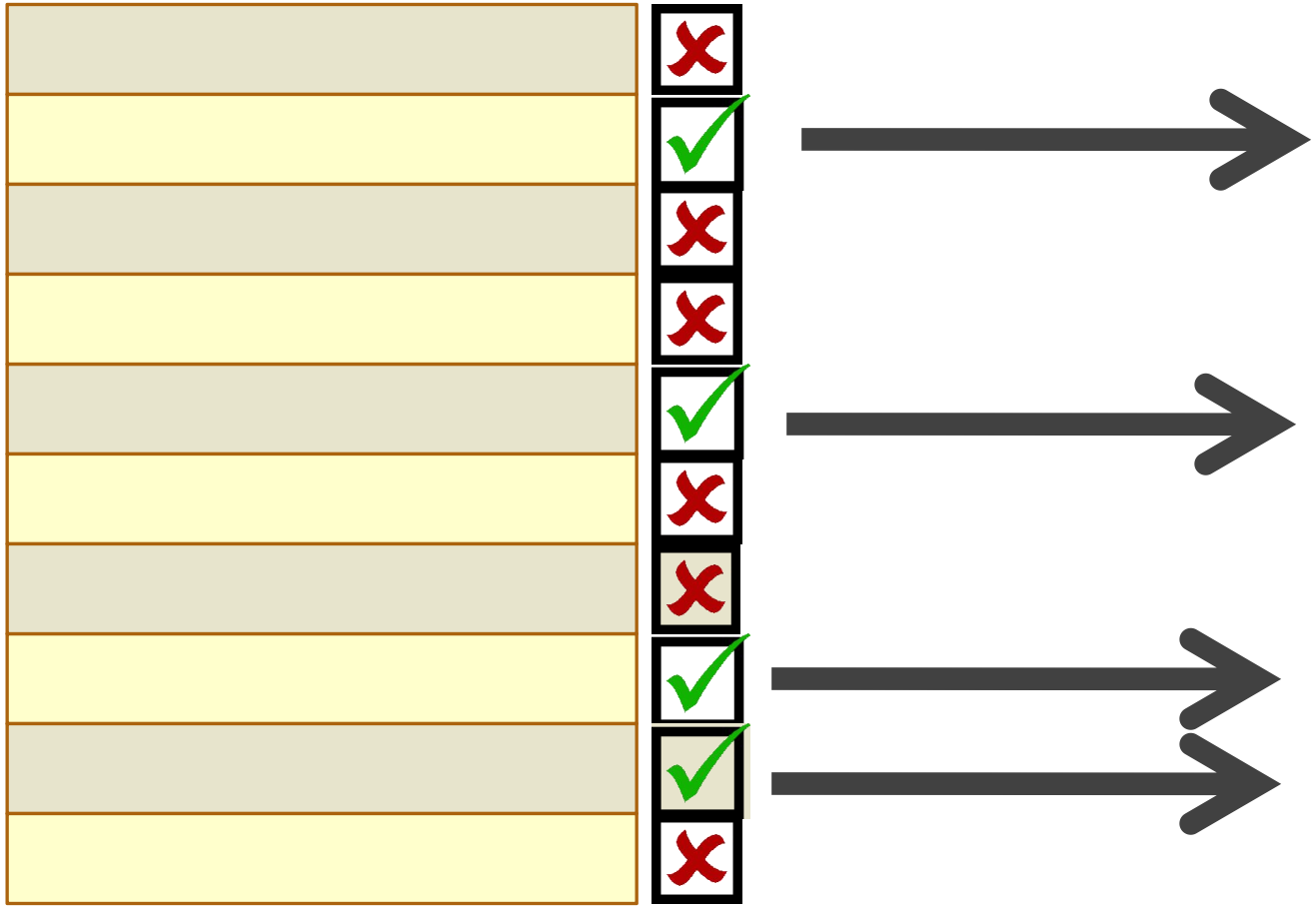
country	number_of_movies
fr	571
ke	1
si	1
eg	11
nz	23
bg	4
ru	153
gh	1
pe	4
hr	1
sg	5
mx	59
cn	200
ee	1
sp	72
cl	14
ec	1
cz	28
dk	30
vn	2
ro	12
mn	1
gb	783
se	59
tw	33
ie	17
ph	42
ar	38
th	21

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

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

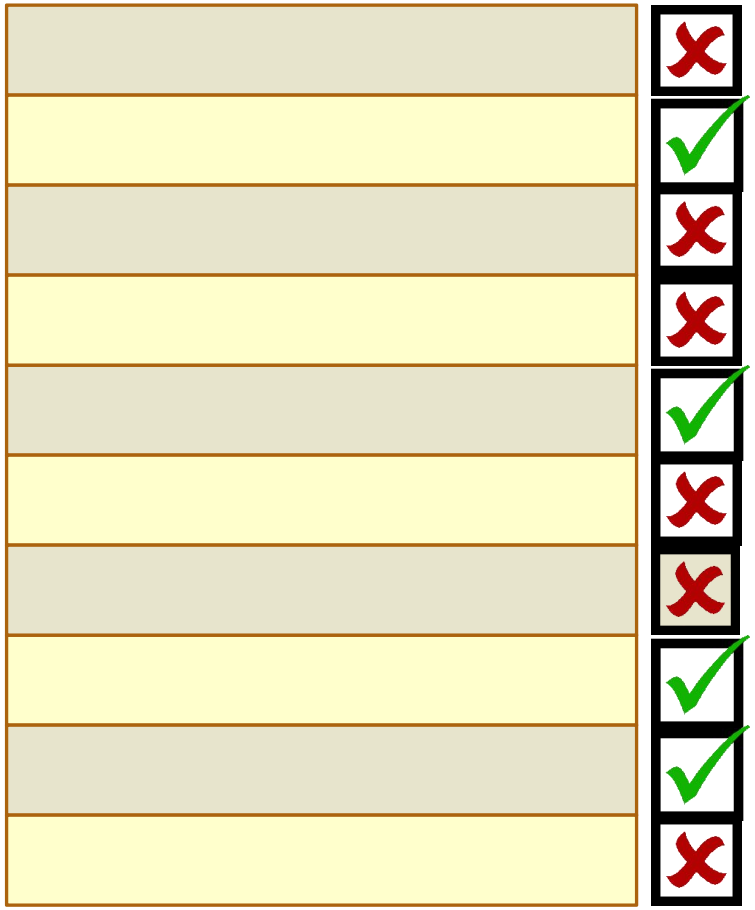
You can also group on several columns. Every column that isn't an aggregate function and appears after SELECT must also appear after GROUP BY.

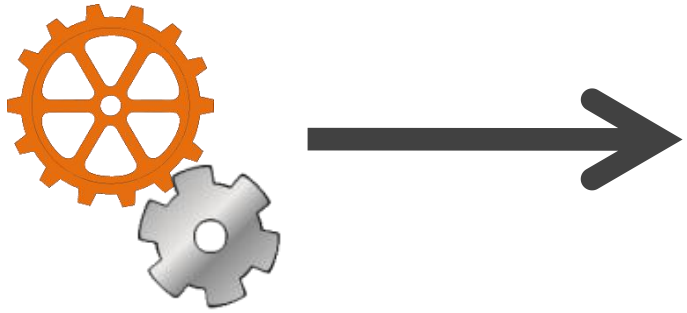
where



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)

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.

country	oldest_movie
fr	1896
ke	2008
si	2000
eg	1949
nz	1981
bg	1967
ru	1924
gh	2012
pe	2004
hr	1970
sg	2002
mx	1933
cn	1913
ee	2007
sp	1933
cl	1926
ec	1999
cz	1949
dk	1910
vn	1992
ro	1964
mn	2007
gb	1916
se	1913
tw	1971
ie	1970
ph	1975
ar	1945
th	1971

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).

country	oldest_movie
fr	1896
ru	1924
mx	1933
cn	1913
sp	1933
cl	1926
dk	1910
gb	1916
se	1913
ca	1933
hu	1918
jp	1926
us	1907
be	1926
at	1925
br	1931
de	1919
au	1906
in	1932
it	1917
ge	1930

(21 rows)

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

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.

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)

SORT

Time complexity of sorting algorithms

O(n*log(n))

Flickr: Randy Robertson

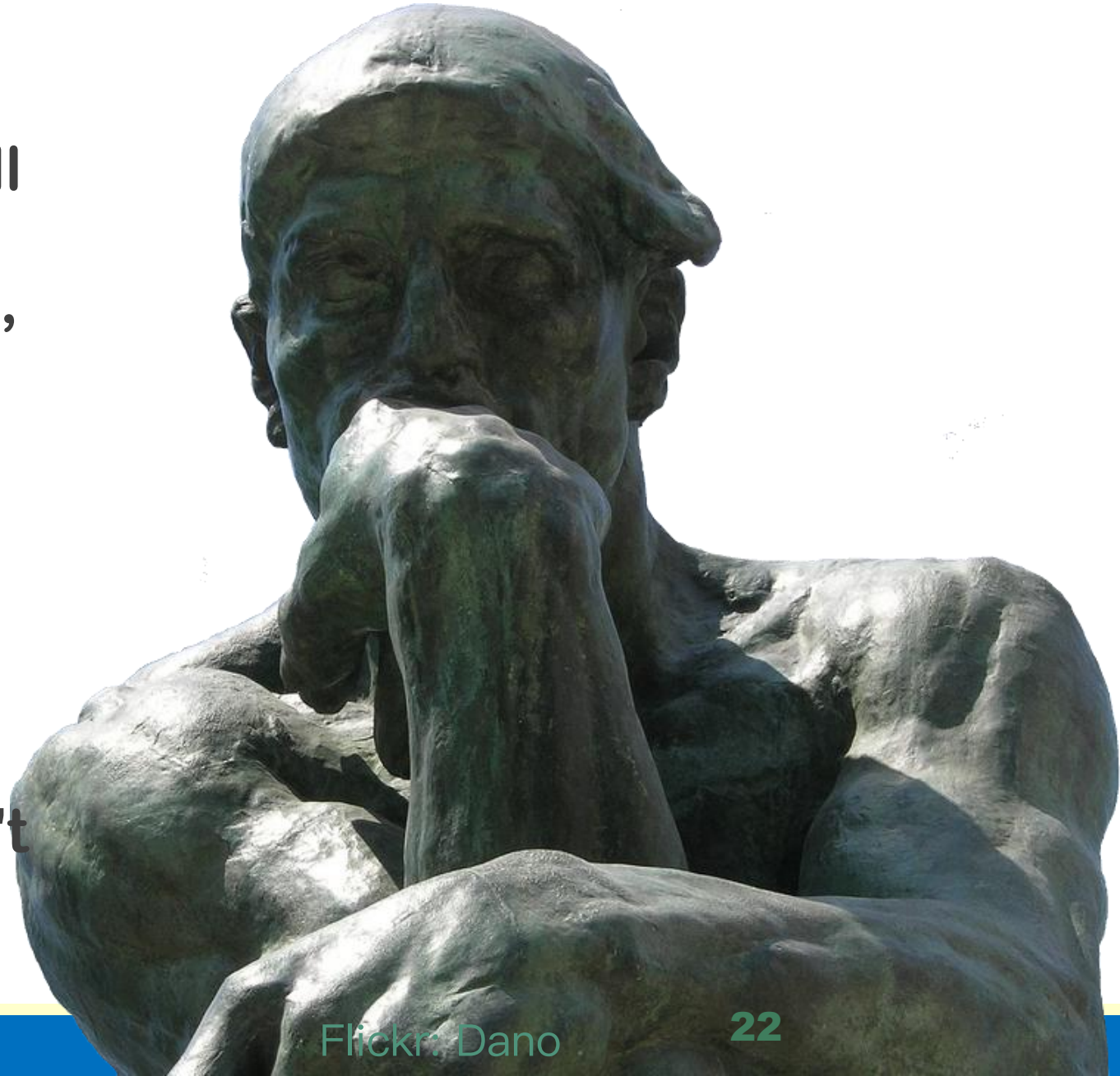
The following query is perfectly valid in SQL. What you are doing is aggregating movies for all countries, then discarding everything that isn't American:

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

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

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.



Flickr: Dano

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

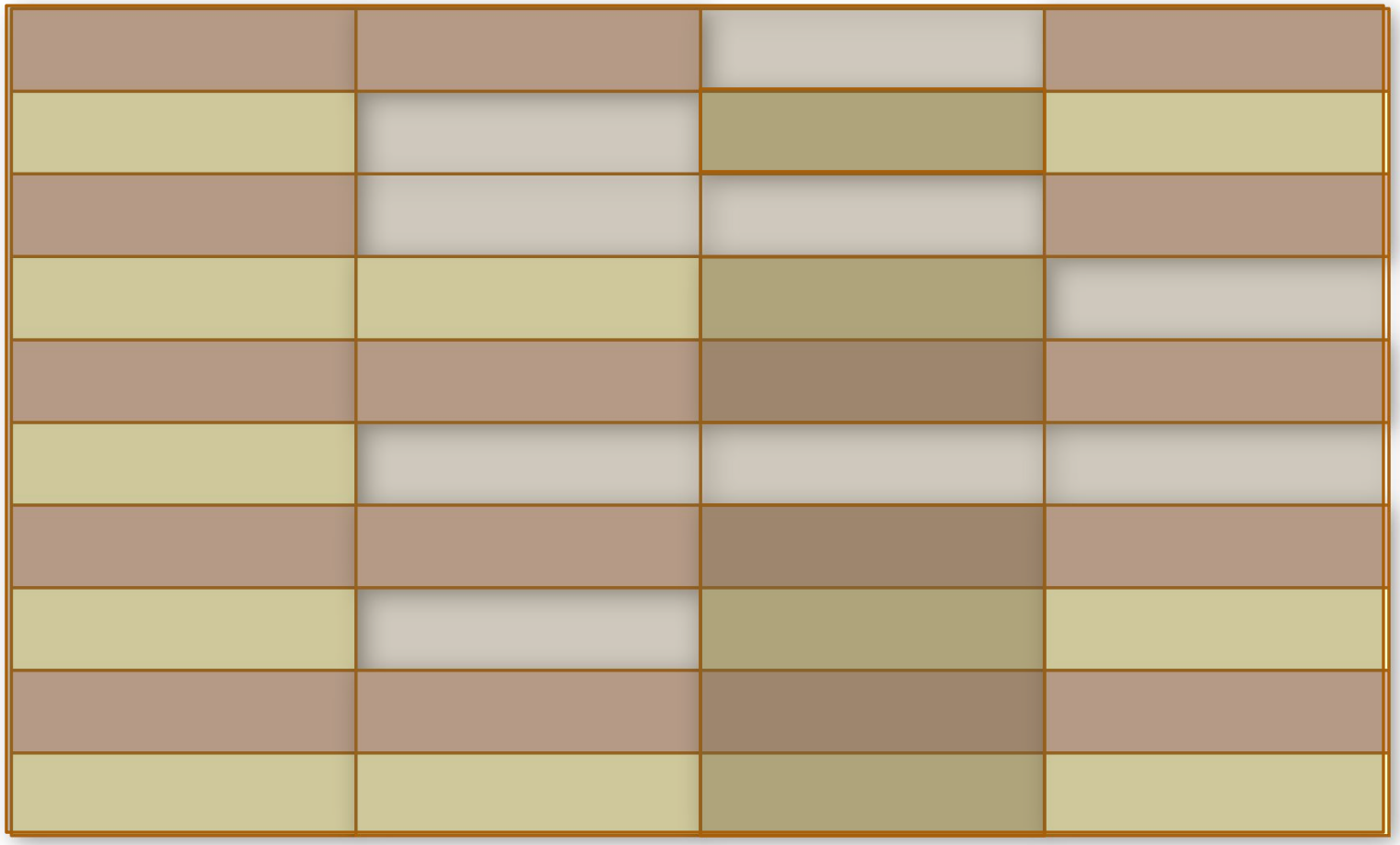
FLickr: Linda Åslund

```
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
```

```
 people_count | birth_year_count | death_year_count
--------------+------------------+------------------
        16489 |            16489 |             5653
(1 row)
```

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.

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

These two queries are equivalent

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

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

How many people
are both
actors and directors?

credits

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
136	378	D
136	433	A
136	434	A
136	435	A
15	38	A
15	359	D
15	360	A
...		


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

There is no restriction such as "that have played in a movie 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')
```

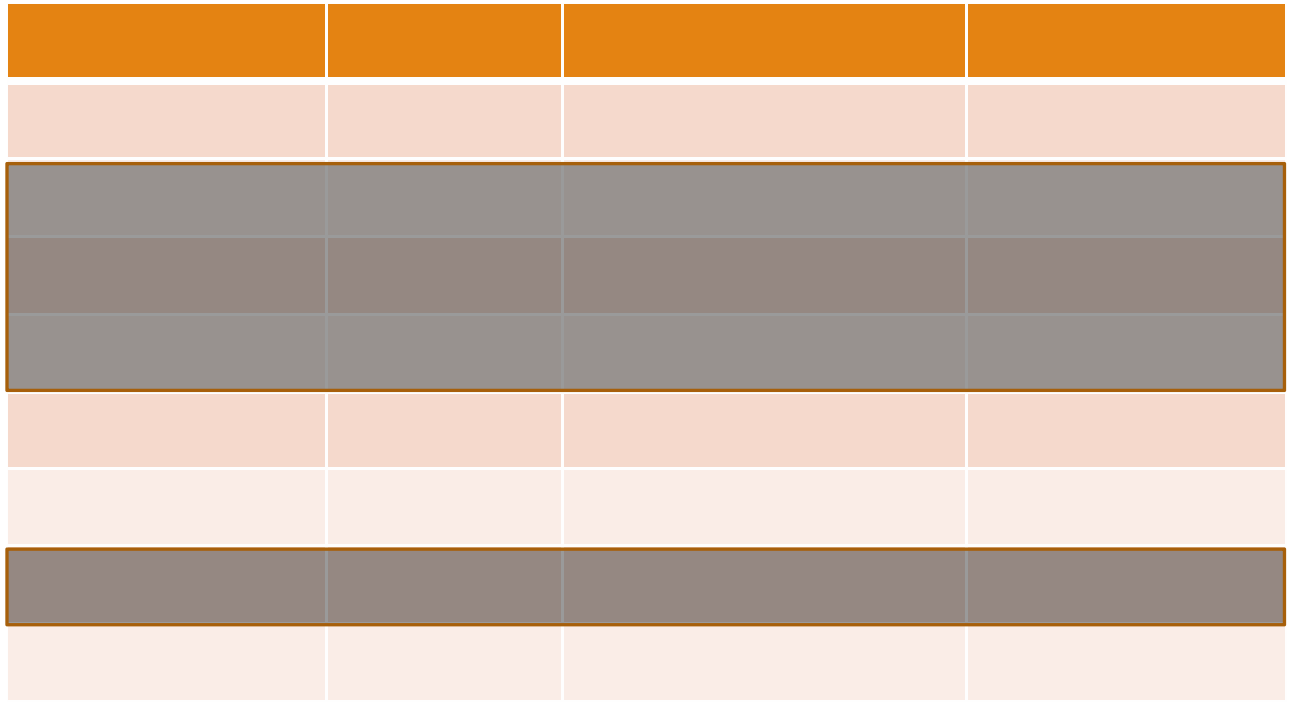
DISTINCT will remove duplicates and provide a true relation. I specify the values for CREDITED_AS because there are no other values now but you can't predict the future (someday there may be producers or directors of photography).

```
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
```

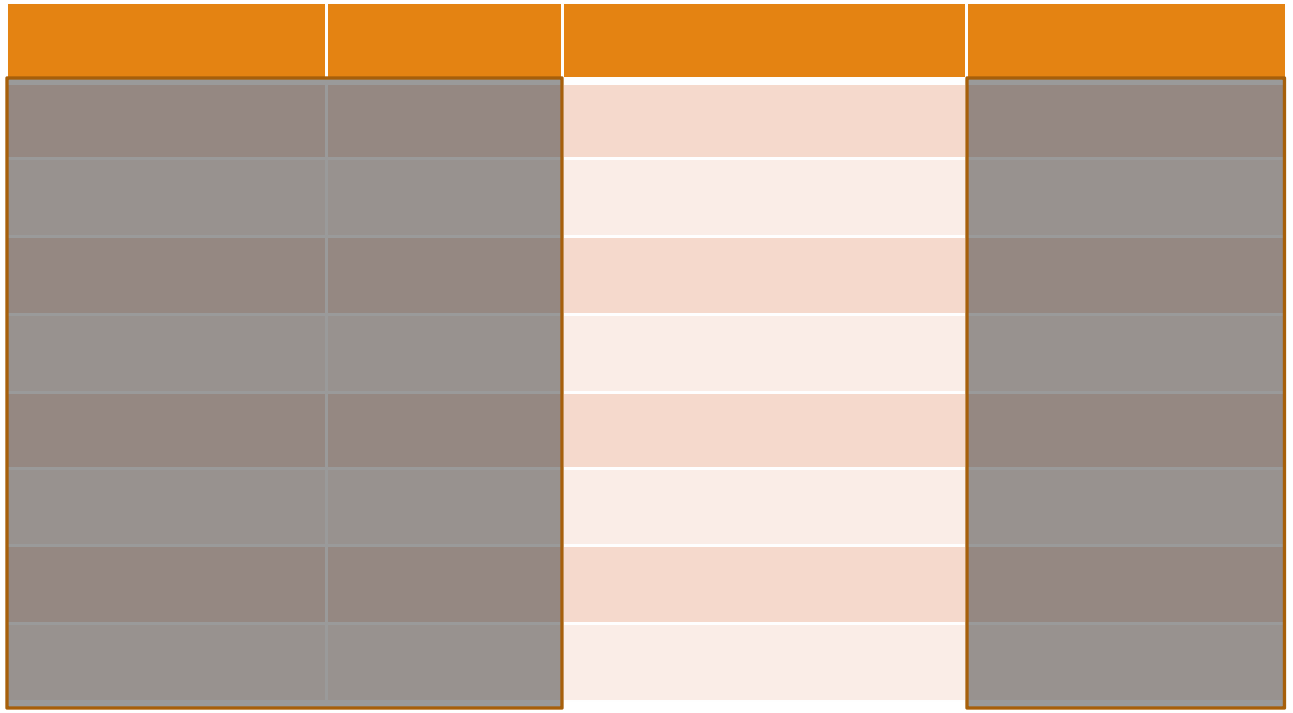
The HAVING selects only people who appear twice … and we just have to count them. Mission accomplished.

4.3 Retrieving Data from Multiple Tables

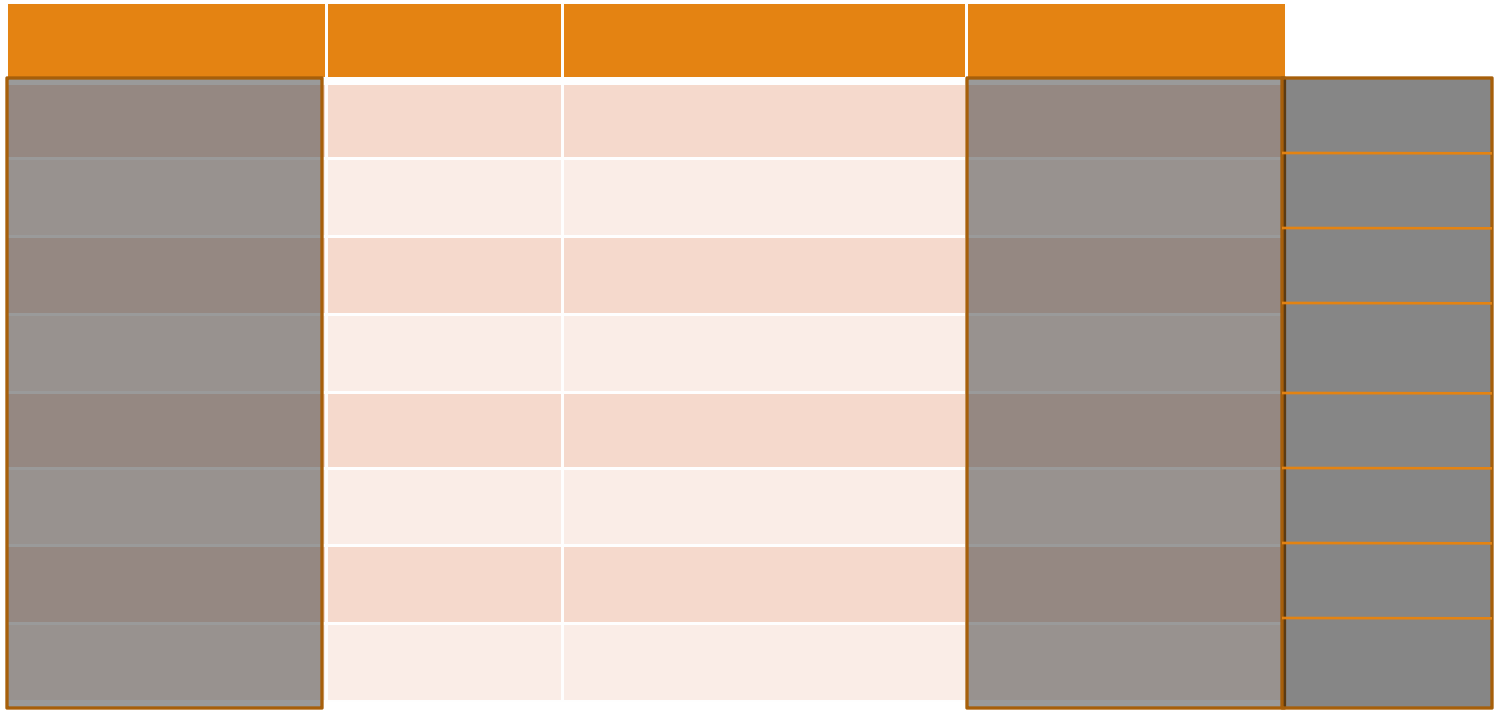
Most contents are from Stéphane Faroult's slides



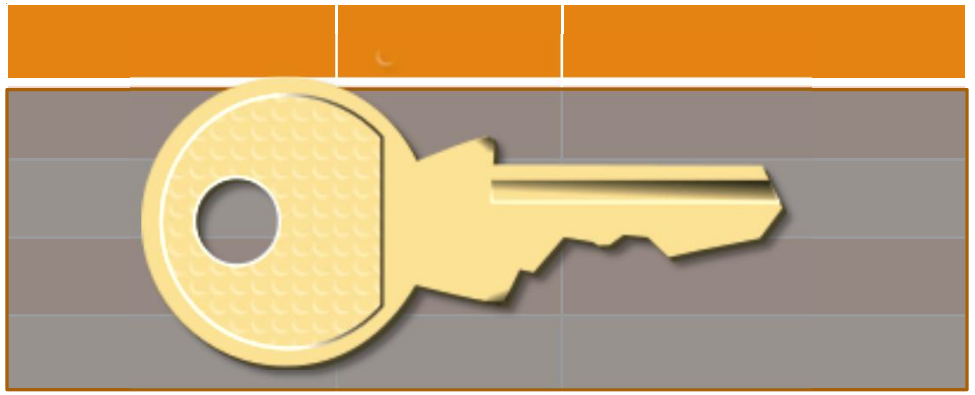
We have seen the basic operation consisting in filtering rows (an operator called SELECT by Codd)



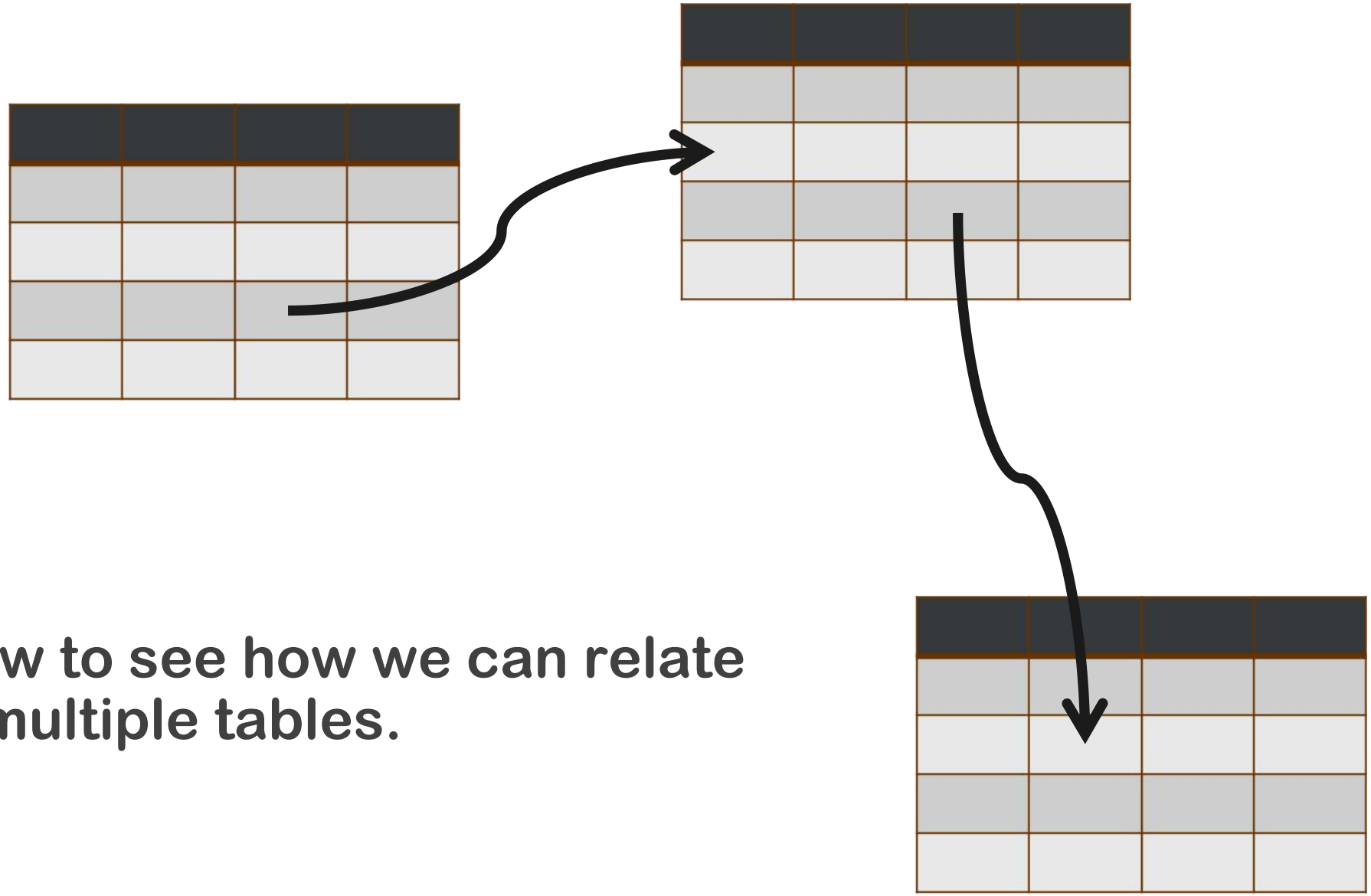
We have seen how we can only return some columns (called PROJECT by Codd), and that we must be careful not to return duplicates when we aren't returning a full key.



We have also seen how we can return data that doesn't exist as such in tables by applying functions to columns.



What is REALLY important is that in all cases our result set looks like a clean table, with no duplicates and a column (or combination of columns) that could be used as a key. If this is the case, we are safe. This must be true at every stage in a complex query built by successive layers.



It's time now to see how we can relate data from multiple tables.

Constraints

This operation is known as JOIN. We have already seen a way to relate tables: foreign key constraints.

movies

movieid	title	country	year_released
1	Casablanca	us	1942
2	Goodfellas	us	1990
3	Bronenosets Potyomkin	ru	1925
4	Blade Runner	us	1982
5	Annie Hall	us	1977
6	Da Nao Tian Gong	cn	1965
7	Sholay	in	1975
8	On The Waterfront	us	1954
9	Lawrence Of Arabia	gb	1962
10	The Third Man	gb	1949
11	Ladri di biciclette	it	1948

The COUNTRY column in MOVIES can be used to retrieve the country name from COUNTRIES

countries

country_code	country_name	continent
ru	Russia	EUROPE
us	United States	AMERICA
in	India	ASIA
gb	United Kingdom	EUROPE
fr	France	EUROPE
cn	China	ASIA
it	Italy	EUROPE
ca	Canada	AMERICA
au	Australia	OCEANIA

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

title	country_name	year_released
12 stulyev	Russia	1971
Al-mummia	Egypt	1969
Ali Zaoua, prince de la rue	Morocco	2000
Apariencias	Argentina	2000
Ardh Satya	India	1983
Armaan	India	2003
Armaan	Pakistan	1966
Babettes gæstebud	Denmark	1987
Banshun	Japan	1949
Bidaya wa Nihaya	Egypt	1960
Variety	United States	2008
Bon Cop, Bad Cop	Canada	2006
Brilliantovaja ruka	Russia	1969
C'est arrivé près de chez vous	Belgium	1992
Carlota Joaquina - Princesa do Brasil	Brazil	1995
Cicak-man	Malaysia	2006
Da Nao Tian Gong	China	1965
Das indische Grabmal	Germany	1959
Das Leben der Anderen	Germany	2006
Den store gavtyv	Denmark	1956

This is done with this type of query. We retrieve, and display as a single set, pieces of data coming from two different tables.

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.

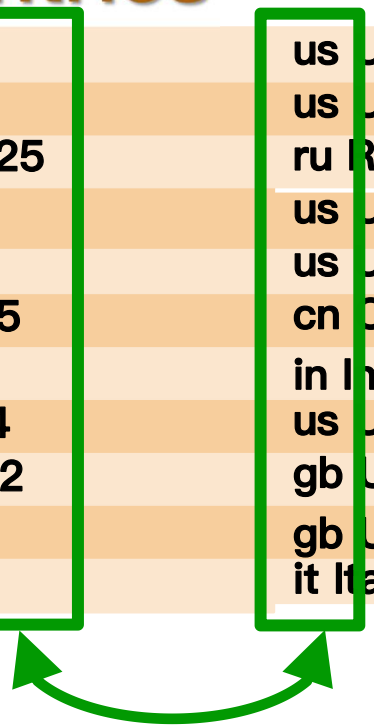
If Table1 has R1 rows, and Table2 has R2, the huge virtual table has R1xR2 rows.

```
select 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	us	United States	AMERICA
2	Goodfellas	us	1990	us	United States	AMERICA
3	Bronenosets Potyomkin	ru	1925	ru	Russia	EUROPE
4	Blade Runner	us	1982	us	United States	AMERICA
5	Annie Hall	us	1977	us	United States	AMERICA
6	Da Nao Tian Gong	cn	1965	cn	China	ASIA
7	Sholay	in	1975	in	India	ASIA
8	On The Waterfront	us	1954	us	United States	AMERICA
9	Lawrence Of Arabia	gb	1962	gb	United Kingdom	EUROPE
10	The Third Man	gb	1949	gb	United Kingdom	EUROPE
11	Ladri di biciclette	it	1948	it	Italy	EUROPE



We use on country_code=country to filter out unrelated rows to make a much smaller virtual talbe.

movies joined to countries

1	Casablanca	us	1942	us	United States	AMERICA
2	Goodfellas	us	1990	us	United States	AMERICA
3	Bronenosets Potyomkin	ru	1925	ru	Russia	EUROPE
4	Blade Runner	us	1982	us	United States	AMERICA
5	Annie Hall	us	1977	us	United States	AMERICA
6	Da Nao Tian Gong	cn	1965	cn	China	ASIA
7	Sholay	in	1975	in	India	ASIA
8	On The Waterfront	us	1954	us	United States	AMERICA
9	Lawrence Of Arabia	gb	1962	gb	United Kingdom	EUROPE
10	The Third Man	gb	1949	gb	United Kingdom	EUROPE
11	Ladri di biciclette	it	1948	it	Italy	EUROPE

```
select title,
       country_name,
       year_released
from   movies
     join      countries
       on country_code = country
where country_code <> 'us'
```

From this virtual table we can retrieve some columns, and apply filtering conditions to any column. As long as there are no duplicates, it's a relation ...

```
join ...
 on column1_from_table1 = column5_from_table2
and column2_from_table1 = column1_from_table2
```

We can join on more than one column, it happens fairly often. Although it's far more frequent to use equality in joins, we can also use other comparison operators, especially when we are joining on several columns.

people

peopleid	first_name	surname	born	died
5	Claude	Rains	1889	1967
10	Lung	Ti	1946	
15	Carol	Reed	1906	1976
20	Ramesh	Sippy	1947	
25	David	Lean	1908	1991
30	Ray	Liotta	1954	
35	Rutger	Hauer	1944	

Now keep in mind the structures of PEOPLE and CREDITS. They are related through a column called PEOPLEID in both tables.

credits

movieid	peopleid	credited_as
1	5	A
6	10	A
10	15	D
7	20	D
9	25	D
137	25	D
2	30	A
4	35	A

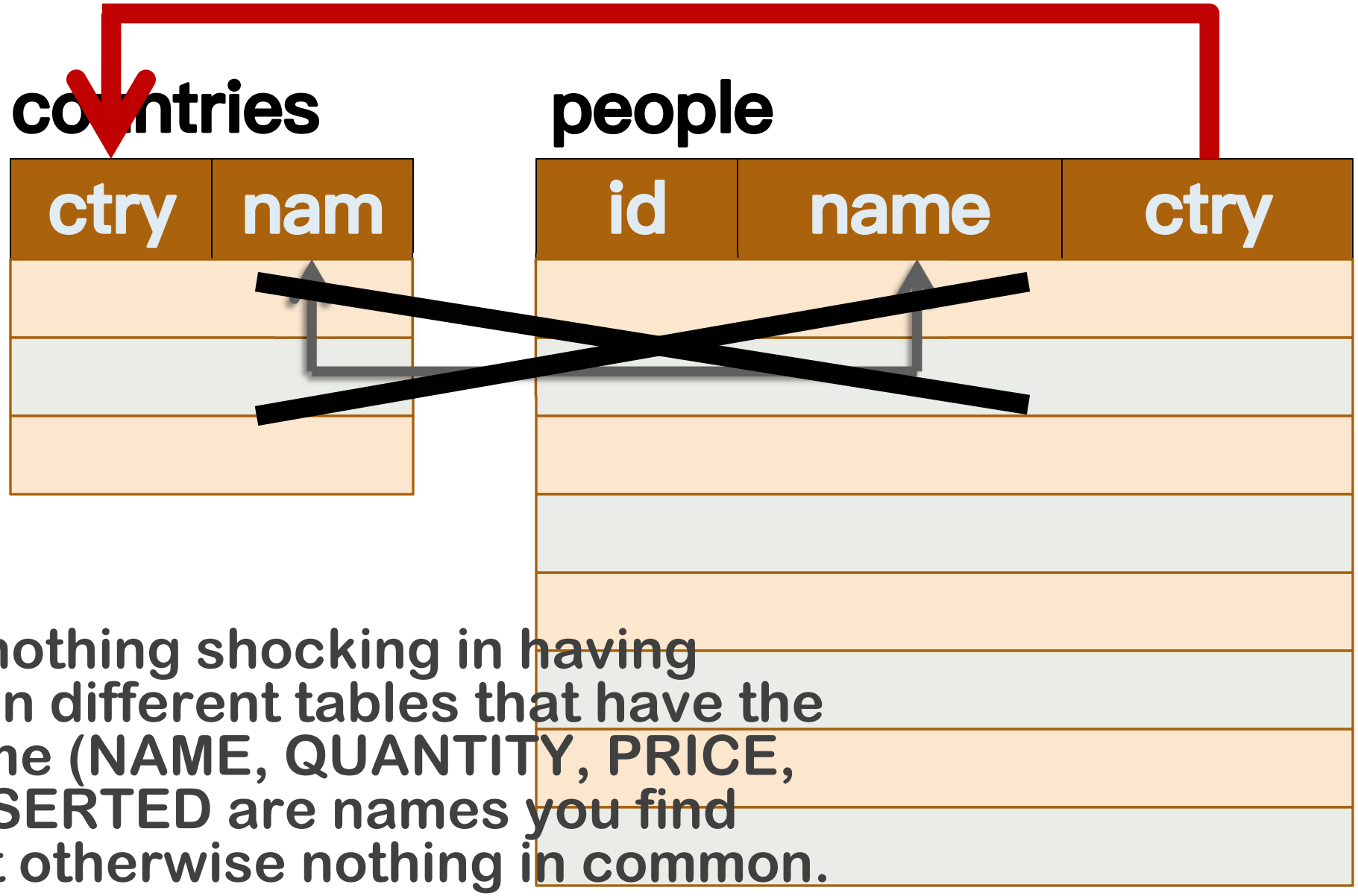
First name and surname of all directors in the database?

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

credits

movieid	peopleid	credited_as
1	5	A
6	10	A
10	15	D
7	20	D
9	25	D
137	25	D
2	30	A
4	35	A

If the name is the same, the matching condition becomes ambiguous. There is something called NATURAL JOIN (unsupported by SQL Server) that basically says "**if a column has the same name, then we should join on it**". Bad idea, because it's purely based on NAMES, and not on foreign keys (which would make sense)



There is nothing shocking in having columns in different tables that have the same name (NAME, QUANTITY, PRICE, DATE_INSERTED are names you find often) but otherwise nothing in common.

```
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.

```
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     as p
     join credits              as c
          on c.peopleid = p.peopleid
where credited_as = 'D'
```

We can give a very short alias to every table in the query (specified after the table name) and use aliases to eliminate ambiguity (side note: most products accept 'people AS p' instead of 'people p', except Oracle that starts abusing you. However, Oracle accepts AS before a COLUMN alias. Go figure).

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

Bonus feature with aliases: as they are short, you can even prefix every column in the query with the alias for the table it comes from even if they are unambiguous. It provides some welcome documentation. We are only seeing two-table joins here, but joining five tables or more is frequent (remember than databases with a few hundred tables are common) and it helps see where every piece of information is sourced from.

4.4 More about Join

Most contents are from Stéphane Faroult's slides

peopleid	first_name	surname...	fatherid	motherid
876	MICHAEL	REDGRAVE		
932	RACHEL	KEMPSON		
1234	VANESSA	REDGRAVE	876	932

A simple example of self join is if, for actor families, each row were containing the identifiers of the father and mother if they are in the database. You can display child and father.

```
select c.first_name || ' ' || c.surname as person,
       f.first_name || ' ' || f.surname as father
from people as c       -- child
     join people as f -- father
       on f.peopleid = c.fatherid
```

One instance of PEOPLE is PEOPLE as a table that only contain children, and the other one as a table that only contains fathers.

```
select ...
from (    [join operation]
    join ...
)x
```

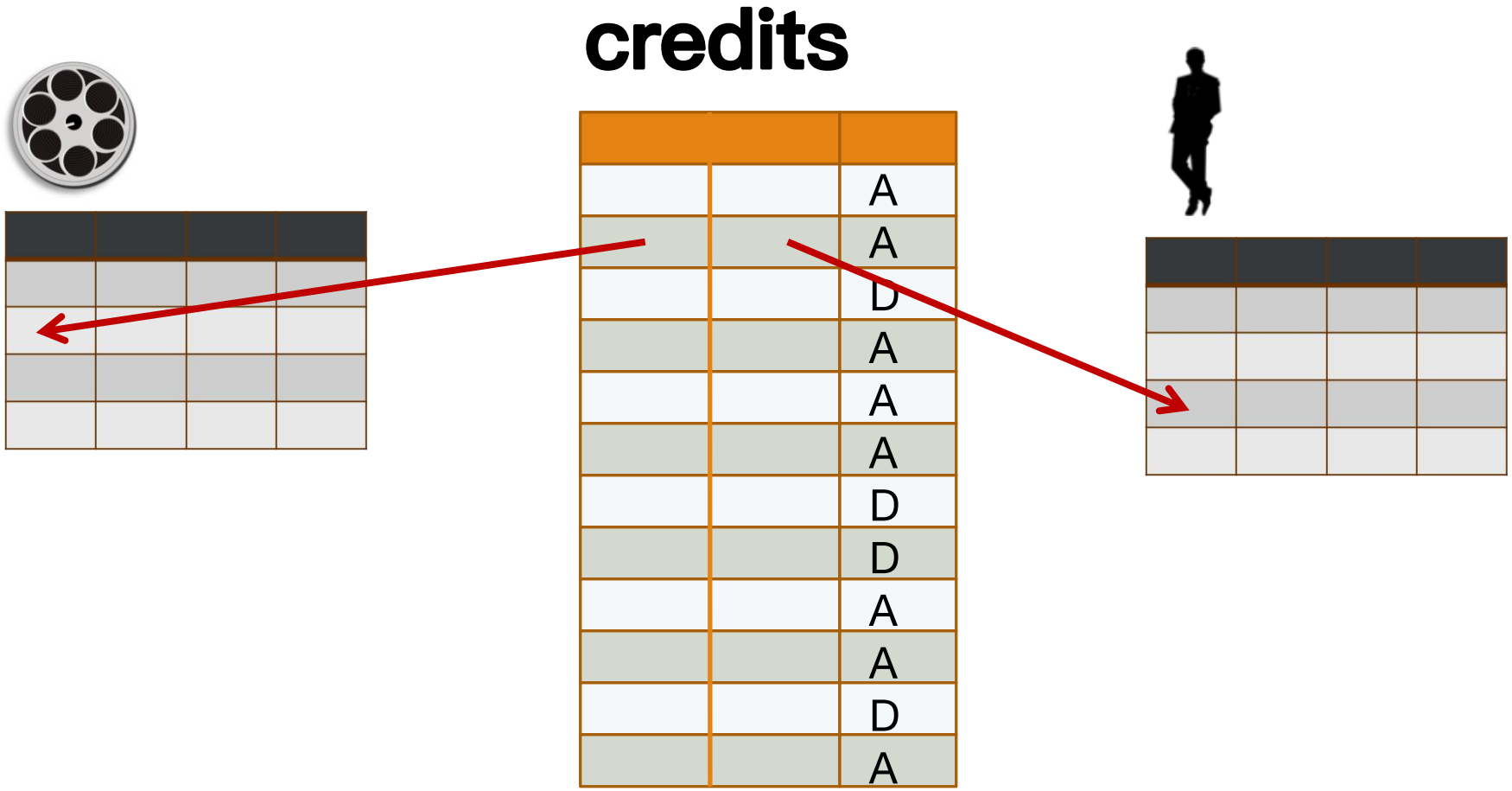
A join can as well be applied to a subquery seen as a virtual table, as long as the result of this subquery is a valid relation in Codd's sense. And if the result of a join is a valid relation, then we can join it again ...

```
select ...
from table1
    join table2
        on ...
    ...
    join tablen
        on ...
```

We can also chain joins the same way we chain filtering conditions with AND. Joins between 10 or 15 tables aren't uncommon, and queries generated by programs often do much worse.

British movie titles with director names?

Let's write a relatively simple query. As you will see, even a simple query can let the door opened to problems.



Finding the tables involved is simple enough.

```
select m.title, p.surname
```

Trouble begins as soon as we start writing column names after SELECT. The title comes from MOVIES, and the surname from PEOPLE. But is it a key?

P.C. Barua

He wasn't a British director, but the legendary Assamese Indian director P.C. Barua directed at least three versions of a very classic Bengali book, "Devdas", in different Indian languages and with different actors.

देवदास 1936

দেবদাস 1937

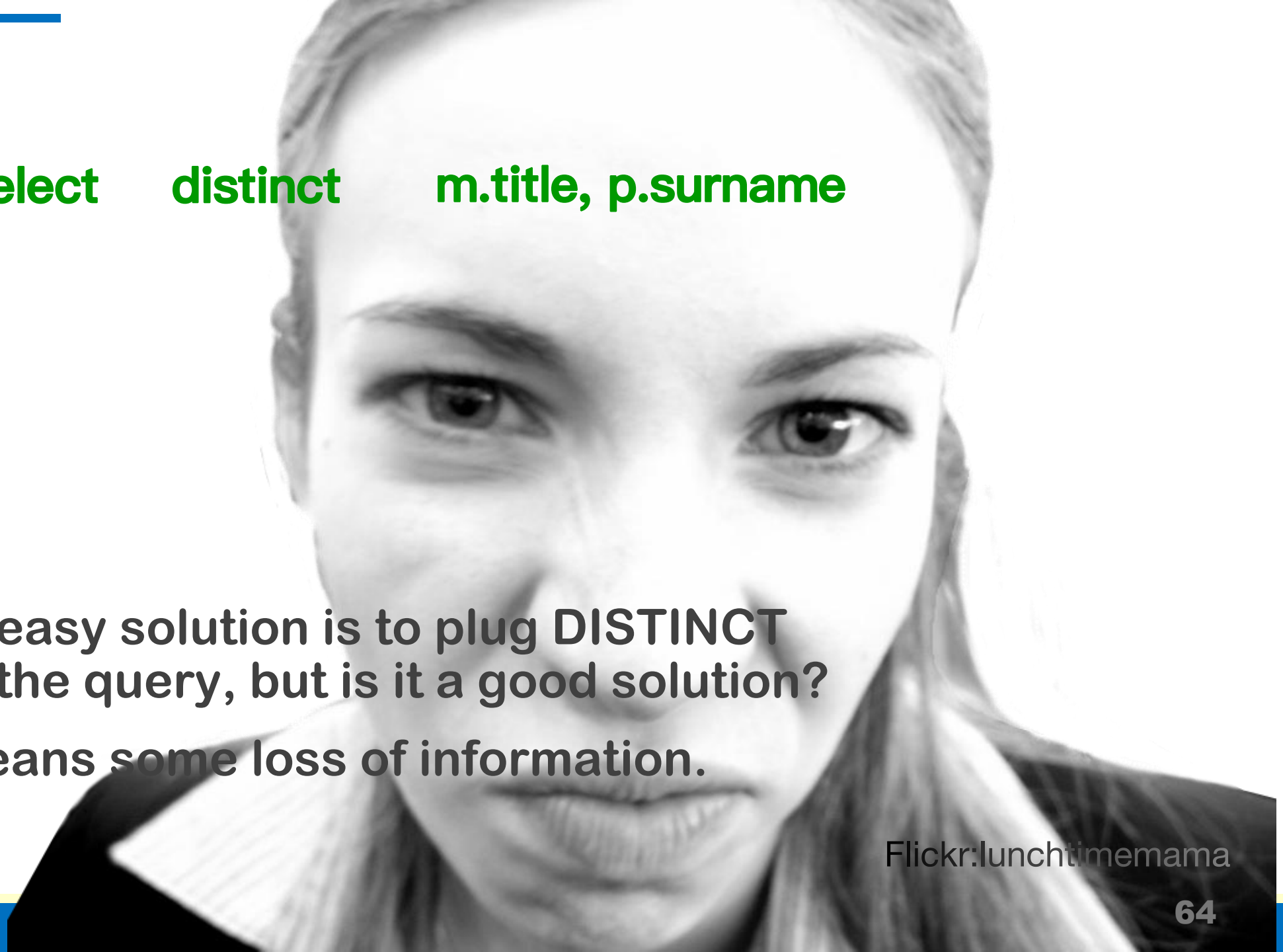
দেবদাস 1935

surname ?

They aren't any more British, but when two brothers co-direct a film, then title and surname are definitely not unique.

How to handle the situation of a movie with two directors?

```
select distinct m.title, p.surname
```



The easy solution is to plug DISTINCT into the query, but is it a good solution?

It means some loss of information.

Flickr:lunchtimemama

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

A better solution is probably to return everything that is required to be certain about uniqueness.

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

One important thing is that the order of tables, even if MOVIES looks prominent here, is completely irrelevant.

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

We could start with PEOPLE or even CREDITS. I have briefly mentioned the optimizer already, it's free to start with any table it wants (it depends on filtering criteria; better to start with the table for which we can select efficiently fewer rows before starting joining

In fact, the JOIN notation was introduced in the late 1990s. The original way from 1974 SQL (still perfectly valid, and still very much in use) is to have a comma-separated list of tables after FROM, and join conditions in the WHERE clause. It's clearer with the original syntax that the order of tables doesn't really matter.

```
select m.year_released, m.title,
       p.first_name, p.surname
from movies m,
     credits c,
     people p
where c.movieid = m.movieid
  and p.peopleid = c.peopleid
  and c.credited_as = 'D'
  and m.country = 'gb'
```

The newer syntax was designed to help differentiate between join conditions (after ON) and plain filtering conditions, and make more difficult to forget a join condition and get a Cartesian product, which is the combination of every row in a table with every row in another table.

```
select m.year_released, m.title,
       p.first_name, p.surname
from movies m,
     credits c,
     people p
```

If you forget WHERE, the SQL engine will not report an error to this statement. It will cause huge amount of rows selected (Rm x Rc x Pc)