

# CS213

# Principles of Database Systems(H)

## Chapter 5

---

Shiqi YU 于仕琪

yusq@sustech.edu.cn

Most contents are from Stéphane Faroult's slides

# 5.1 Outer Join

---

Shiqi Yu 于仕琪

yusq@sustech.edu.cn

```
select m.year_released, m.title,  
       p.first_name, p.surname  
from movies m  
       join credits c  
           on c.movieid = m.movieid  
       join people p  
           on p.peopleid = c.peopleid  
where c.credited_as = 'D'  
      and m.country = 'us'  
      and year_released = 2018
```

Look for the directors of  
American movies  
released in 2018 ...

year_released	title	first_name	surname
2018	Red Sparrow	Francis	Lawrence
2018	Ready Player One	Steven	Spielberg
2018	A Star Is Born	Bradley	Cooper
2018	Mary Queen of Scots	Josie	Rourke

(4 rows)

```
select m.movieid, m.year_released, m.title  
from movies m  
where m.country = 'us'  
and year_released = 2018
```

... which is that all movies weren't listed in the previous query. We have for instance 'Black Panther' in the list returned by the query above, and not in the previous one.

movieid	year_released	title
8987	2018	Red Sparrow
8988	2018	Ready Player One
8990	2018	A Star Is Born
8992	2018	Mary Queen of Scots
9202	2018	Black Panther
9203	2018	A Wrinkle in Time

(6 rows)

# movies


# people

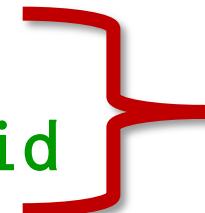

# credits




The reason is that its director isn't credited for it in the database. No credits, no link with PEOPLE, the movie vanishes.

```
filmdb=# select * from credits where movieid=9202;
movieid | peopleid | credited_as
-----+-----+-----
 9202 |      5588 | A
 9202 |     15870 | A
 9202 |      3933 | A
(3 rows)
```

```
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 = 'us'  
  and year_released = 2018
```



It's easier to understand why with the traditional SQL way of writing joins. Join conditions can also be interpreted as filtering conditions. If we can't find a row in CREDITS, then the condition isn't true.



Flickr: John Wigham

Once again, we have to decide on what we want: are we primarily interested by movies for which the director is known, or by all 2018 American movies?

# outer join inner join

If we want to see all 2018 American movies in the database,  
we need to resort to an extended kind of join called an  
**OUTER join** (the regular join is often called **INNER join**)

	✓			
	✓			
	✓			
	✓			

The diagram shows a left table with 5 columns and 5 rows, and a right table with 2 columns and 4 rows. The left table has green checkmarks in its first column. The right table has blue boxes. The result is a 5x6 table where the last column is empty. The first four rows have blue boxes in the last two columns, while the fifth row has a blue box in the first column and an empty last column.

An outer join tells to complete the result set with NULLs if we can't find a match in the outer joined table.

---

# 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.

Country Name	Number of movies
--------------	------------------



Let's take a simpler example than the three-table join between MOVIES, CREDITS and PEOPLE and let's just try to count how many movies we have per country. We may have no movies from smaller countries, or newer countries that haven't produced one movie yet.

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

# Display zero when we have no movies from a country?

Sometimes with a LEFT OUTER JOIN we don't want to see NULL. NULL is fine with text information (such as a director name) but for quantitative information such as a number of movies we'd rather see zero. This is easy to do.

```
select c.country_name,  
       case  
           when x.number_of_movies is null then 0  
           else x.number_of_movies  
       end 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
```

We can use a CASE construct.

country_name	number_of_movies
Algeria	2
Angola	0
Benin	0
Botswana	0
Burkina Faso	2
Burundi	0
Cameroon	0
Central African Republic	0
Chad	0
Comoros	0
Congo Brazzaville	0
Congo Kinshasa	0
Cote d'Ivoire	0
Djibouti	0
Egypt	11
Equatorial Guinea	0
Eritrea	0
Ethiopia	0
Gabon	0
Gambia	0
Ghana	1
Guinea	0
Guinea-Bissau	1
Kenya	1
Lesotho	0
Liberia	0

```
select c.country_name,  
       coalesce(x.number_of_movies, 0)  
                           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
```

Or we can use the more concise COALESCE() function (available with all products) that takes an indeterminate number of parameters and returns the first one that isn't NULL.

```
from table1 1  
left outer join table2 2  
on
```



Once again, beware that **CONTRARY TO WHAT HAPPENS WITH THE ORDINARY (inner) JOIN, ORDER IS IMPORTANT.**  
If you want to see all entries from table1, table1 must come first with a left outer join.

```
select count(*)  
from movies m  
    inner join countries c  
    on m.country = c.country_code
```

```
select count(*)  
from movies m  
    left outer join countries c  
    on m.country = c.country_code
```

```
select count(*)  
from movies m  
    left outer join countries c  
    on m.country <> c.country_code
```

What is the returned number?

Why?

- TRUE
- FALSE
- NULL

# British movie titles with director names when available?

But there are other traps with outer joins. Let's try to answer this question again. From the way it is worded, we want to see all British movies, so MOVIES will be the leading table.

# Which table will be fully shown?

Where can we miss information? Only in CREDITS. Because of referential integrity, every row in CREDITS will have a matching row both in MOVIES and in PEOPLE.

Miss?



What we may miss are rows in CREDITS, which must be outer joined

```
select m.year_released,  
       m.title, m.country,  
       p.first_name, p.surname,  
       c.credited_as  
  from movies m  
  left outer join credits c  
    on c.movieid = m.movieid  
 inner join people p  
   on p.peopleid = c.peopleid
```

Will our query be correct then? Not yet.

```
from movies m 1  
left outer join credits c 2  
on c.movieid = m.movieid  
inner join people p 3  
on p.peopleid = c.peopleid
```

This condition will fail if we return a NULL row from CREDITS.

movies

credits

people

movies	credits	people
Unknown director		Oooooops

A diagram illustrating a database query issue. It shows three tables: 'movies', 'credits', and 'people'. The 'movies' table has a row for an 'Unknown director'. The 'credits' table has a row for this movie, but its 'people' column is null. The 'people' table has no row for this person. A red arrow points to the 'people' column in the 'credits' table, and another red arrow points to the 'people' table with the word 'Oooooops'.

```
select a.year_released, a.title,  
      a.first_name, a.surname  
from (select m.year_released,  
          m.title, m.country,  
          p.first_name, p.surname,  
          c.credited_as  
    from movies m  
    left outer join credits c  
      on c.movieid = m.movieid  
    left outer join people p  
      on p.peopleid = c.peopleid  
) a  
where a.credited_as = 'D' ← outer join killer  
and a.country = 'gb' But if the left outer join returns NULL, then  
CREDITED_AS cannot be 'D' and the movie will  
disappear.
```

we need a second outer  
join to return NULL from  
PEOPLE when there is  
no row in CREDITS.

```
select a.year_released, a.title,  
      a.first_name, a.surname  
from (select m.year_released,  
          m.title, m.country,  
          p.first_name, p.surname,  
          c.credited_as  
    from movies m  
      left outer join credits c  
        on c.movieid = m.movieid  
      left outer join people p  
        on p.peopleid = c.peopleid  
) a  
where a.credited_as = 'D'( or a.credited_as is null)  
and a.country = 'gb'
```



?



Movies

People

movieid      peopleid      A

Credits

What is vicious with the previous wrong query is that it will correctly display a movie as long as we know neither director nor actors. As soon as we know one actor (but not the director) the movie will vanish again.

```
select a.year_released, a.title,  
      a.first_name, a.surname  
from (select m.year_released,  
          m.title, m.country,  
          p.first_name, p.surname,  
          c.credited_as → A  
       from movies m  
       left outer join credits c  
         on c.movieid = m.movieid  
       left outer join people p  
         on p.peopleid = c.peopleid  
) a  
where a.credited_as = 'D' ( or a.credited_as is null)  
and a.country = 'gb'
```

Here is why: if we have an actor, the left outer join with CREDITS will behave like an inner join. But it will return 'A' in CREDITED\_AS

```
select m.year_released, m.title,  
      p.first_name, p.surname  
from movies m  
  left outer join credits c  
    on c.movieid = m.movieid  
  left outer join people p  
    on p.peopleid = c.peopleid  
where (c.credited_as = 'D'  
      or c.credited_as is null)  
and m.country = 'gb'
```



Exactly the same thing happens when we haven't all the joins as a subquery, but just a succession of joins followed by a WHERE clause.

---

The problem was this:

**Get movie titles  
and director name if available**

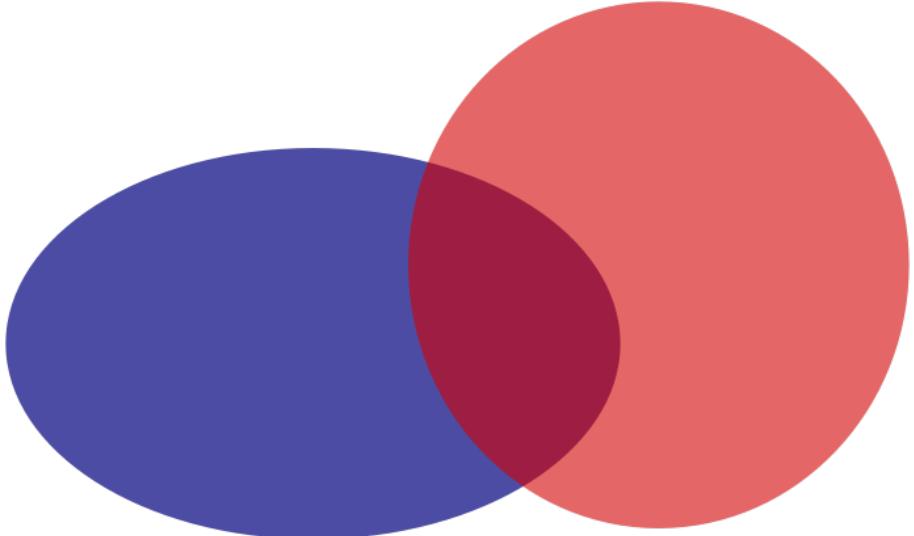
The query does this:

**Get movie titles and people  
involved, then display if director  
known or no people found**

and it's not exactly the same thing. It only is the same thing  
when we know the director, or when we know of nobody  
involved with the movie.

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

This answers exactly the question asked. It's not necessary to have a subquery of British movies for MOVIES, but it is necessary to have a subquery that only returns directors.



## 5.2 Set Operators

---

Shiqi Yu 于仕琪

yusq@sustech.edu.cn

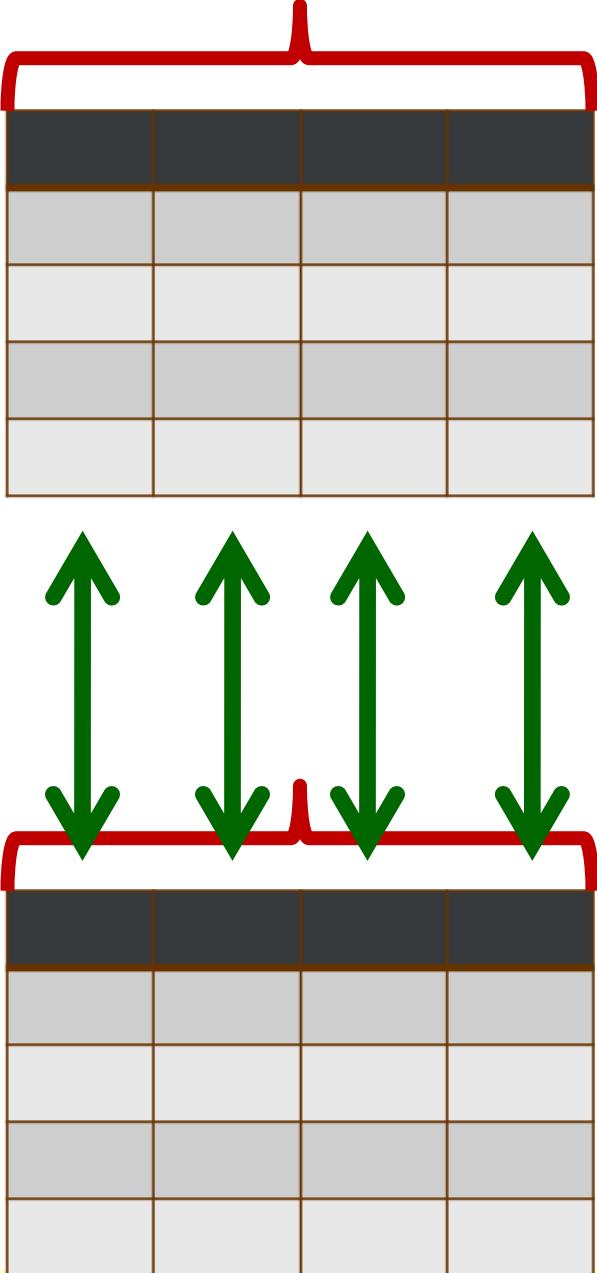
# UNION

The most important (by far) set operator is UNION that takes two result sets and combines them into a single result set.

## US and GB movies, 1940s

```
select movieid, title, year_released, country
from movies
where country = 'us'
    and year_released between 1940 and 1949
union
select movieid, title, year_released, country
from movies
where country = 'gb'
    and year_released between 1940 and 1949
```

For instance we could combine American movies of the 1940s with British movies of the 1940s (OR would be more efficient than two separate searches)



**UNION requires two commonsensical conditions: to combine the results of two queries, they must return the same number of columns, and the data types of corresponding columns must match.**

---

Imagine that you are managing a website that sells subscriptions for movies, with a "standard" subscription and a more expensive "premium" subscription that gives access to more recent movies stored in another table (not necessarily the best of designs, but it's another question)

movies

premium\_movies

You want to display to your "premium" subscribers the content of BOTH tables at once, which you will do with UNION.

```
select 'regular' as class,  
       movieid,  
       title,  
       year_released  
  from movies  
union  
select 'premium' as class,  
       movieid,  
       title,  
       year_released  
  from premium_movies
```

We are going to return with a UNION data from both table. One thing that you need to know is that UNION eliminates duplicates because when you put two duplicate-free relations together, nothing guarantees that you won't have duplicates, except when you have different constants like here.

```
select 'regular' as class,  
       movieid,  
       title,  
       year_released  
  from movies  
union all  
select 'premium' as class,  
       movieid,  
       title,  
       year_released  
  from premium_movies
```

When you know that you CANNOT have duplicates, then you don't need to go through the step of duplicate removal, which is costly. In that case, instead of saying UNION, you say UNION ALL. UNION ALL doesn't mean that you want duplicates, it means that you know that there cannot be any duplicates between the two queries.

# Last year's views plus year-to-date views

Sometimes you NEED to add ALL to UNION. Suppose that you have two tables, one that stores all the views of your movies last year (for reference) and one with all the views for the current year, and you want to sum them both.

```
select x.movieid,  
       sum(x.view_count) as view_count  
  from  
( select movieid,          Goodfellas  
        sum(view_count) as view_count  
   from last_year_data  
  group by movieid  
 union  
 select movieid,          Goodfellas  
        sum(view_count) as view_count  
   from current_year_data  
  group by movieid  
) x  
group by x.movieid
```

It may happen  
that last year's  
count is  
**EXACTLY** this  
year's count.

Goodfellas	2356
------------	------

union

Goodfellas	2356
------------	------

If this is the case, UNION will see a duplicate, and will remove one of the rows. Result, counts will be correct for almost all movies, except this one if this is the only one in this situation, for which it will be half what it should be! Very difficult to notice that the result is wrong.

Goodfellas

2356

union all

Goodfellas

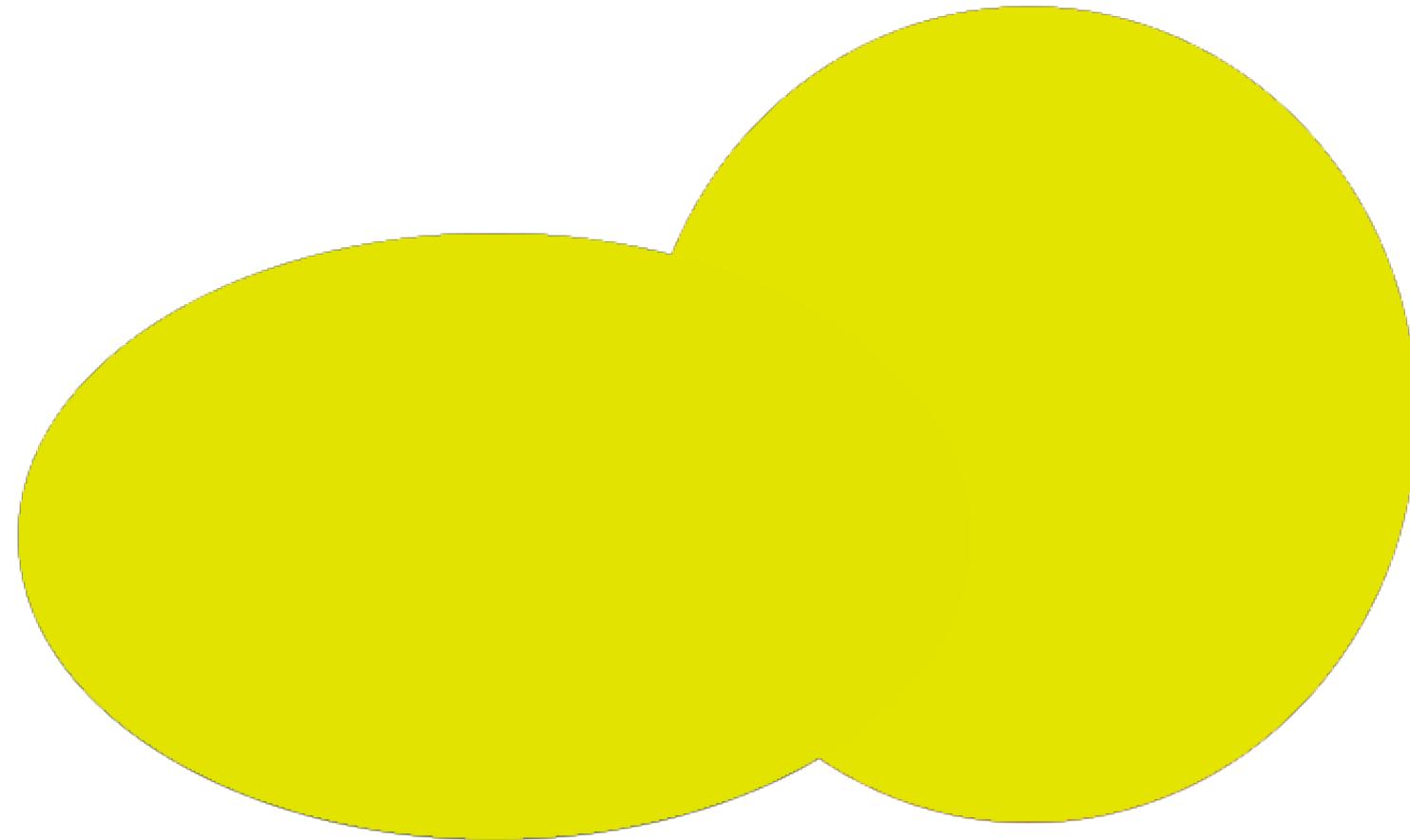
2356

In fact, it may be a "technical duplicate", but it's not a "real duplicate" because both rows represent completely different things, counts for two different years that happen, by mishap, to be identical. We need UNION ALL.

```
select x.movieid,
       sum(x.view_count) as view_count
from (select 'last year' as period,
            movieid,
            sum(view_count) as view_count
         from last_year_data
        group by movieid
      union all
      select 'this year' as period,
            movieid,
            sum(view_count) as view_count
         from current_year_data
        group by movieid) x
   group by x.movieid
```

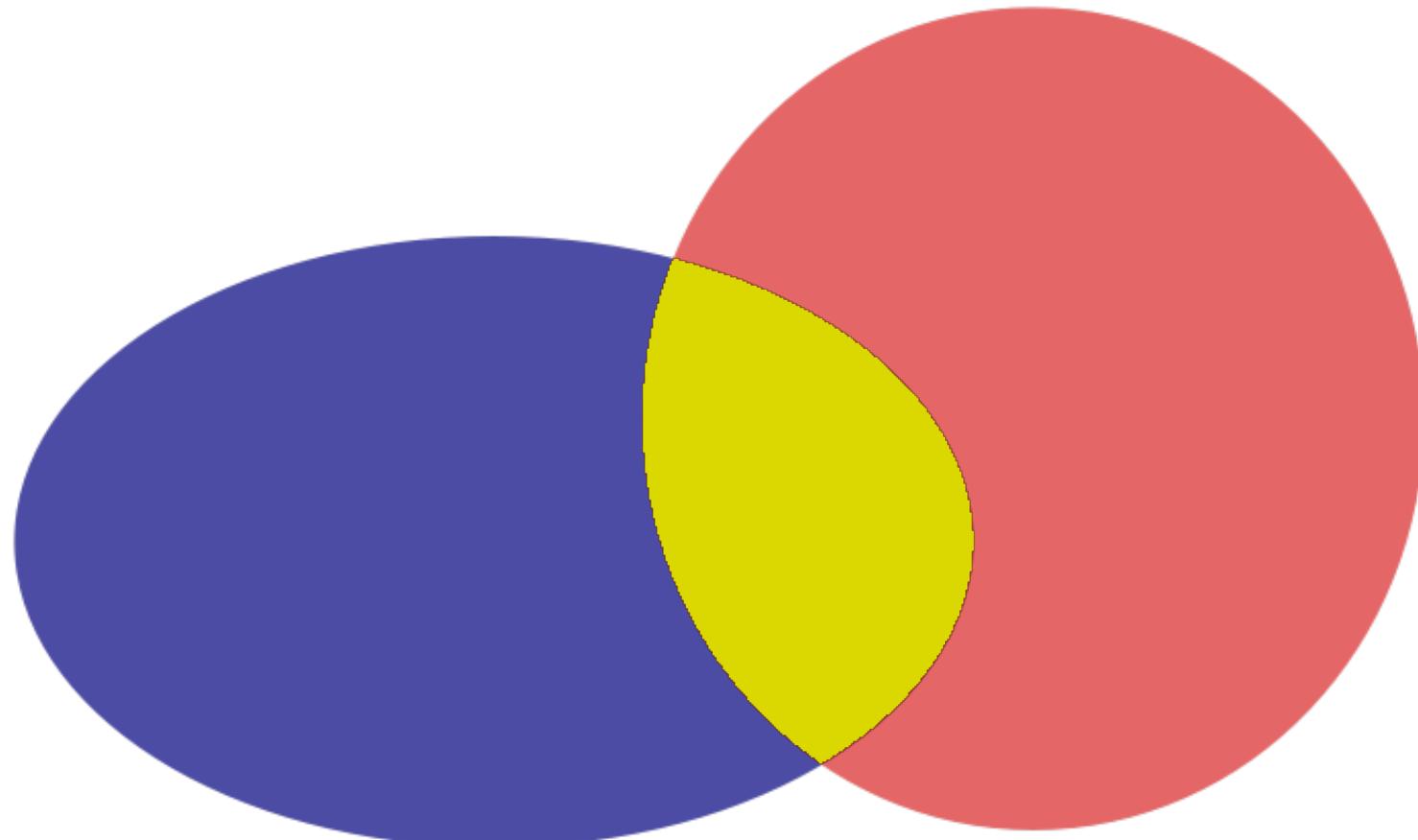
In such a case, I like to add a constant that documents that we are talking about different things and justifies using UNION ALL.

# union



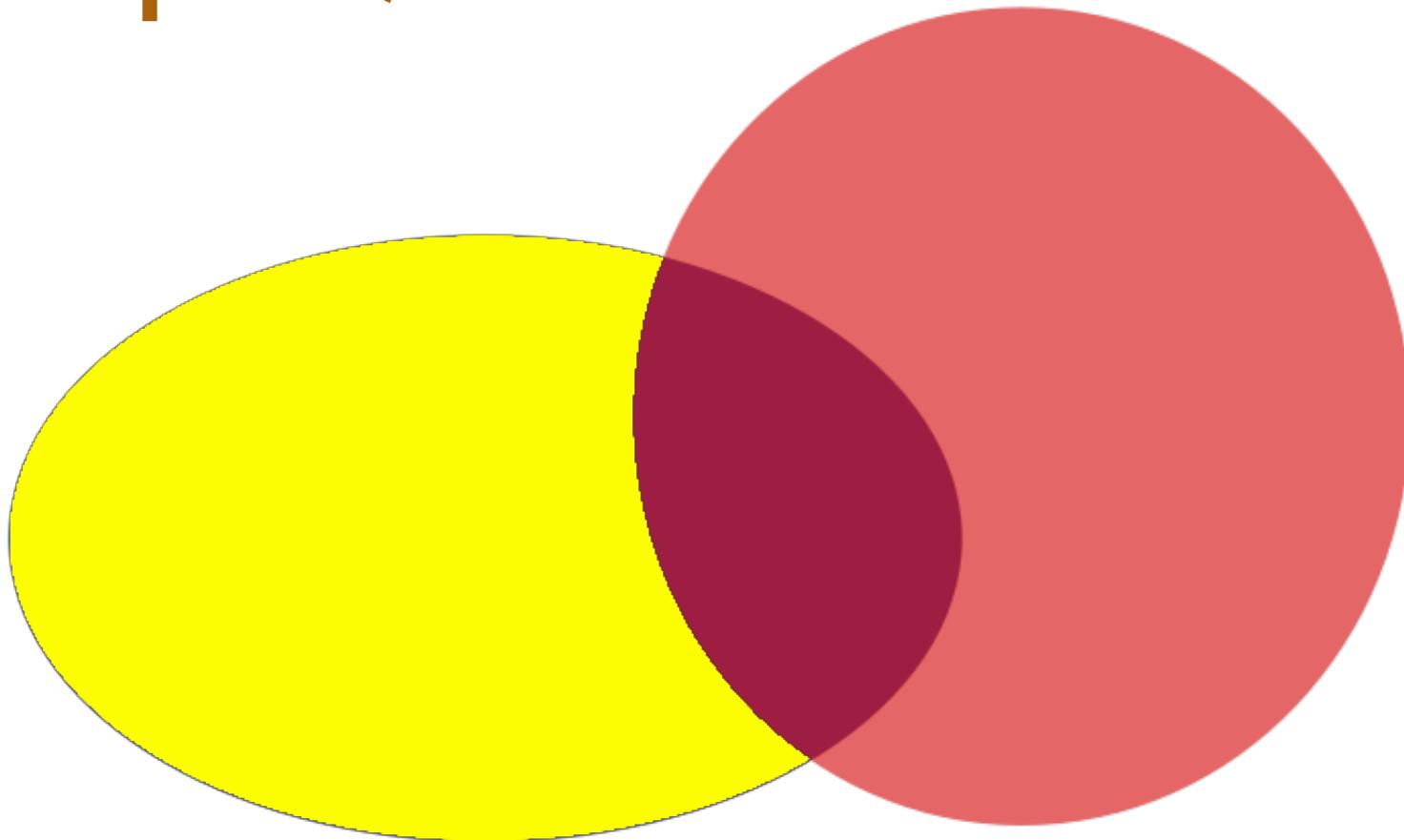
**UNION is the most used set operator. It's not the only one.**

# 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.

intersect → inner join

except → outer join

If they aren't used as much as UNION, it's because contrary to them UNION provides a functionality unavailable otherwise. Finding common rows can be performed with a simple JOIN. Rows present in one set and absent from another can be found with an outer join and an IS NULL condition on a mandatory column of the second table to specify that no match was found.

country codes that are both in  
**movies** and **countries**

Let's take an example and find country codes that are both  
in MOVIES and in COUNTRIES.

```
select country_code  
from countries  
intersect  
select distinct country  
from movies
```

We could use INTERSECT. Like UNION it removes duplicates but it's sounder (and more efficient) to eliminate them as soon as possible with a DISTINCT.

```
select c.country_code  
from countries c  
inner join  
(select distinct country  
from movies) m  
on m.country = c.country_code
```

Or we can join, as the join will eliminate every country code from COUNTRIES that cannot be found in MOVIES.

---

# Different problem:

Countries for which we haven't any movie.

Finding countries for which we have no movie, though, requires both COUNTRIES and MOVIES. There are two ways to answer the question.

or

minus

select country\_code

from countries

except

select distinct country

from movies

```
select c.country_code
from countries c
    left outer join
        (select distinct country
         from movies) m
    on m.country = c.country_code
where m.country is null
```

Or a LEFT OUTER JOIN and a condition IS NULL on a column that cannot be null, which proves that it was returned by the LEFT OUTER JOIN because no match was found.

# Equivalent columns

In practice, the fact that we must have the same number and types of columns with set operators is fairly constraining.



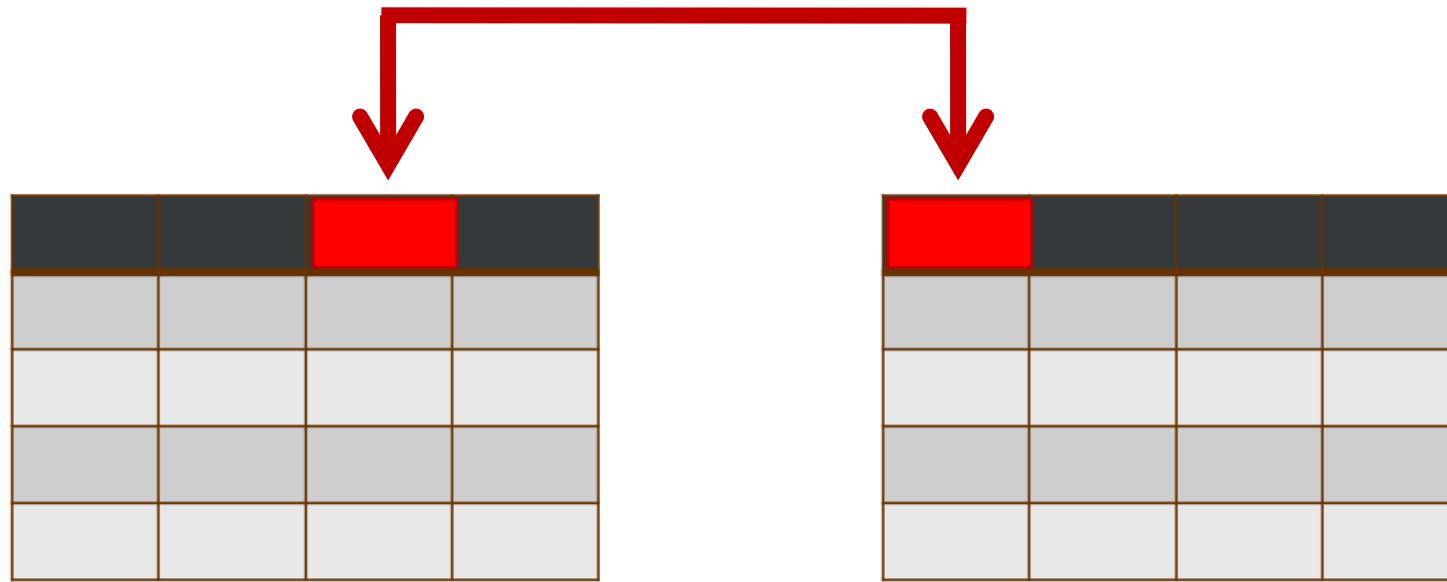
Flickr: Tomáš Obšívač

# 5.3 Subqueries

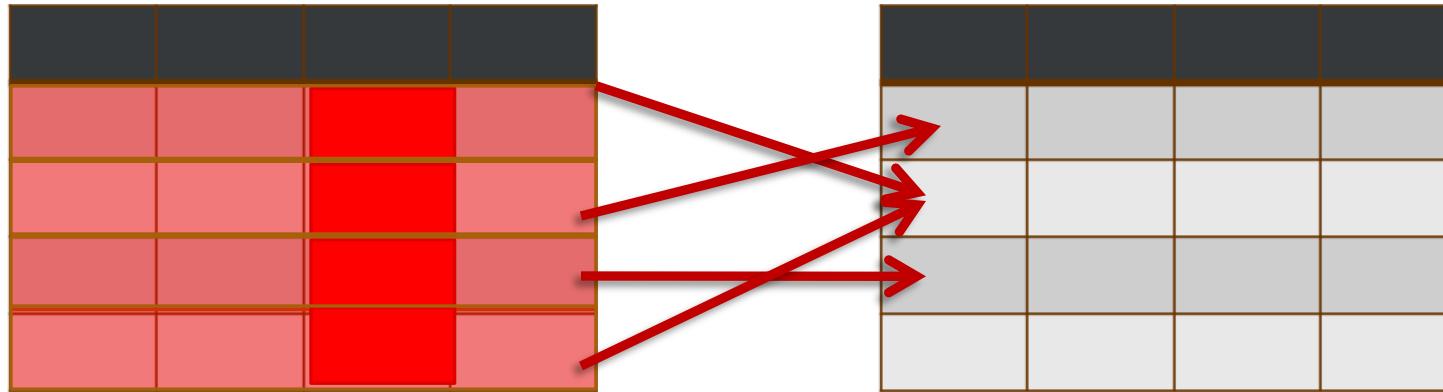
---

Shiqi Yu 于仕琪

yusq@sustech.edu.cn



**When you run a join, you link tables through common values in one (sometimes several) column.**



One way to perform a join, known as a "nested loop" (there are other algorithms) is to scan a table and for each row look for matching values in the column of the other table on which the join is performed.

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

So you can imagine that when you execute this join, for every row in MOVIES the code in column COUNTRY will be checked, and, if it's not 'us', the corresponding movie title, year\_realeased and country name will be fetched from COUNTRIES.

You could also write the query like this, with a subquery after the SELECT.

```
select m.title, m.year_released,  
       (select c.country_name  
        from countries c  
        where c.country_code = m.country)  
              as country_name  
   from movies m  
  where m.country <> 'us'
```

The condition in the subquery is provided by the value in the row from MOVIES that is currently inspected. The subquery is said to be CORRELATED with the outer (main) query.

X		us	
✓		ru	
✓		in	
X		us	
X		us	
✓		gb	
✓		de	
✓		in	

```
select c.country_name  
from countries c  
where c.country_code = 'ru'
```

```
select c.country_name  
from countries c  
where c.country_code = 'in'
```

```
select c.country_name  
from countries c  
where c.country_code = 'gb'
```

```
select c.country_name  
from countries c  
where c.country_code = 'de'
```

```
select c.country_name  
from countries c  
where c.country_code = 'in'
```

The subquery is fired  
for every returned row.

---

Note though that this isn't exactly equivalent to a join.  
What happens if we don't find the country name?  
(which won't happen here because of the foreign key)

```
select m.title,  
       (select c.country_name  
        from countries c  
       where c.country_code = m.country)  
              as country_name  
  from movies m  
 where m.country_code <> 'us'
```

The subquery would return nothing, also known as NULL.

```
select m.title,  
       c.country_name  
  from movies m  
        left outer join countries c  
          on c.country_code = m.country  
 where m.country <> 'us'
```

So, strictly speaking, a subquery after the SELECT is more equivalent to a LEFT OUTER JOIN.

a subquery in the FROM  
cannot be correlated.

**from** clause

a subquery after SELECT  
usually is

**select** list

**where** clause  
a subquery after WHERE  
usually is.

uncorrelated

correlated

uncorrelated  
or correlated

---

```
in ( . . . , . . . , . . . )
```

```
select country, title  
from movies  
where country in ('us', 'gb')  
and year_released between 1940  
                    and 1949
```

IN () is a nice alternative way to replace a series of conditions on the same column linked by OR.

```
in (select col  
     from ...  
     where ...)
```

But IN () is far more powerful than this, because what is between parentheses may be, not only an explicit list, but also an implicit list of values generated by a query.

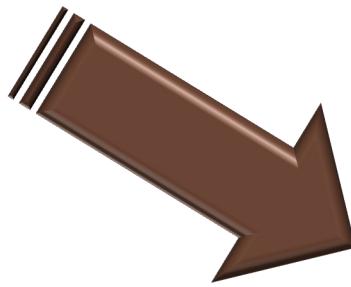
```
select country, year_released, title  
from movies  
where country in  
    (select country_code  
     from countries  
     where continent = 'EUROPE')
```

For instance, this query would return every European movie without having to painfully list all European countries.

```
(col1, col2) in  
  (select col3, col4  
   from t  
   where ...)
```

Some products (Oracle, DB2, PostgreSQL with some twisting)  
even allow comparing a set of column values (the correct word is  
"tuple") to the result of a subquery.

in



distinct

One thing important to know is that IN () means an implicit DISTINCT in the subquery. I always prefer to make the implicit explicit, but some people disagree.

```
select country, year_released, title  
from movies  
where country in  
    (select country_code  
     from countries  
     where continent = 'EUROPE')
```

Actually, such a subquery is, as you see, not correlated: it doesn't depend on the outside query. In fact, the subquery could be moved up in the FROM clause.

```
select m.country, m.year_released, m.title  
from movies m  
inner join  
(select country_code  
from countries  
where continent = 'EUROPE') c  
on c.country_code = m.country
```

Unique?



But if you do so, then the JOIN is a relational operation that should only be performed between two relations – is what I return in my subquery unique? In that case yes, as it's the primary key.

# Demonstrably unique

---

## no distinct

If I am sure that what I return can only be unique (because I return either a primary key or unique constraint, plus possibly other columns) I shouldn't have DISTINCT, which will just add costly and unnecessary processing.

```
select country, title  
from ...  
      inner join  
        (select distinct ...  
         from ...)  
      on ...  
  
where col in  
(select distinct ...  
         from ...)
```

But, if there is the shadow of a possibility, however remote, that one day I might have duplicates (in other words, if I haven't the guarantee of a constraint), then I should have DISTINCT otherwise I may have wrong results with a JOIN (although not with a IN (), but I prefer documenting the possibility of a problem)

**Explicit**  
is better than  
**Implicit**

---

Duplicate rows? Checked.

# duplicate rows?

The other source of problems in SQL is null

# null?

and you probably won't be disappointed.

---

## 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:

$(\text{col} > \text{NULL}) \rightarrow \text{NULL}$

$(\text{col} = \text{NULL}) \rightarrow \text{NULL}$

...

col is NULL  $\rightarrow$  True or  
False

---

## Logical operators

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  $\neq$  'a'  
**and** col  $\neq$  'b'  
**and** col  $\neq$  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 people  
where first_name not in  
(select first_name  
from people  
where born < 1950)
```

```
filmdb=# select *  
from people  
where first_name not in  
(select first_name  
from people  
where born < 1950)  
;  
peopleid | first_name | surname(0 rows)
```

A subquery that returns a NULL in a NOT IN () will always give a null condition, and the result will vanish. That's what happened with Arletty, born long before 1970. If you want to be safe, you should add a condition saying that you DON'T WANT null values if they are possible.

```
select *  
from people  
where first_name not in  
(select first_name  
    from people  
   where born < 1950  
     and first_name is not null)
```

from clause

select list

where clause



So far we have only seen uncorrelated subqueries in the WHERE clause.

---

# exists

Correlated queries in the WHERE clause are used with the (NOT) EXISTS construct.

# not exists

NEVER try to correlate an IN()!

and exists  
(select ...  
...)



In an EXISTS I have, as with subqueries after a SELECT, a reference to a value from **the current row** of the outer query.

```
select distinct m.title  
from movies m  
where exists  
(select null  
from credits c  
inner join people p  
on p.peopleid = c.peopleid  
where c.credited_as = 'A'  
and p.born >= 1970  
and c.movieid = m.movieid)
```

For instance the movies with at least one actor born in 1970 or later.

A black and white close-up photograph of a man's face. He has a wide-eyed, shocked expression, with his mouth slightly open. His hands are raised to his face, fingers near his eyes and nose, as if he is peeking through a hole or reacting to something scary. The lighting is dramatic, with strong highlights and shadows.

And I've used that scary null.

select  
null

```
select m.title  
from movies m  
where exists  
(select 'hello'  
     from credits c  
      inner join people p  
        on p.peopleid = c.peopleid  
   where c.credited_as = 'A'  
     and p.born >= 1970  
     and c.movieid = m.movieid)
```

In fact I could have used anything, and NULL emphasizes it.

**peopleid**

I'm not interested in whom, or when precisely they were born. I just want to check that there is such a person.

# movies

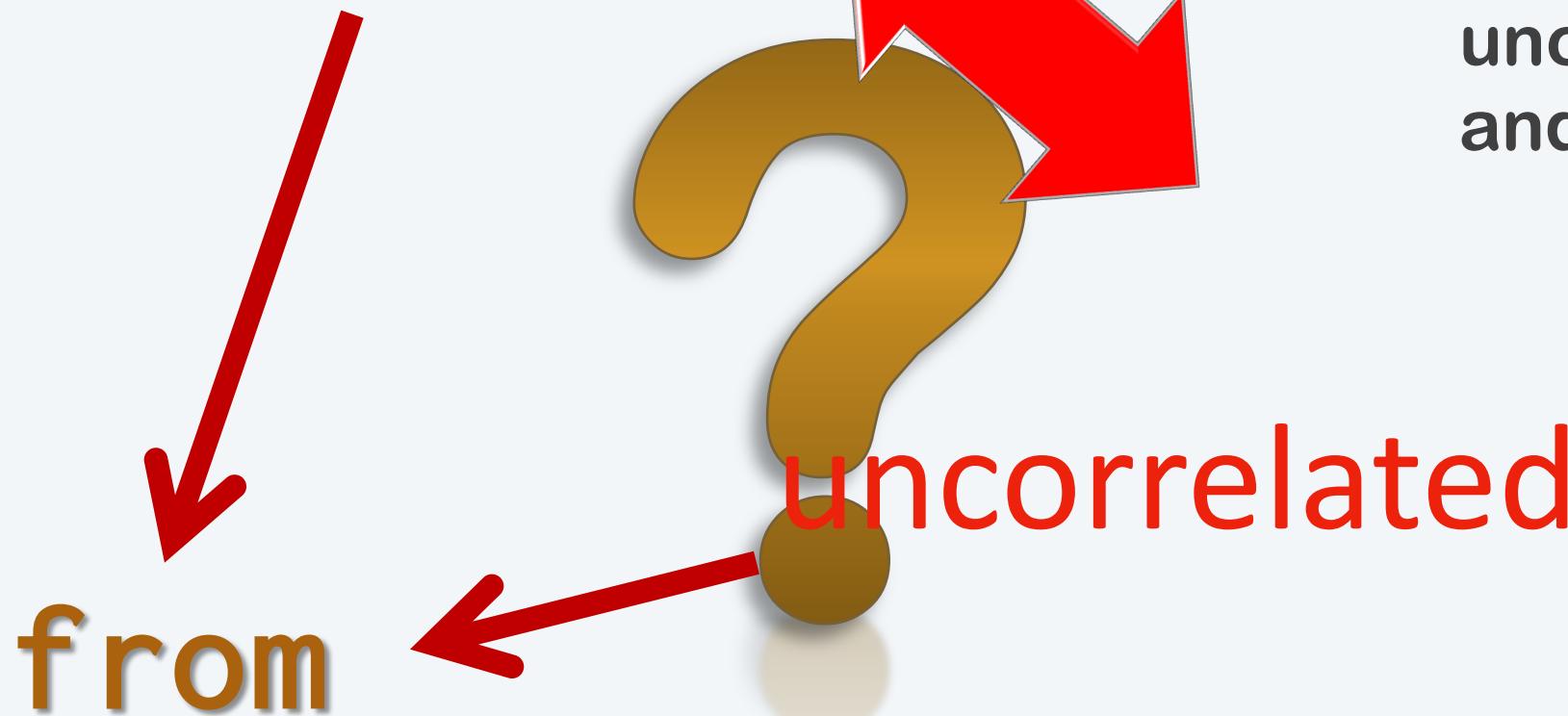
A diagram consisting of a grid of squares. A large red arrow points vertically downwards from the top-left towards the center. From the bottom-right corner, five red lines radiate outwards towards the left.

```
(select null  
from credits c  
      inner join people p  
            on p.peopleid = c.peopleid  
where c.credited_as = 'A'  
      and p.born >= 1970  
      and c.movieid = current movieid)
```

```
select distinct m.title
from movies m
where m.movieid in
  (select distinct c.movieid
   from credits c
    inner join people p
      on p.peopleid = c.peopleid
   where c.credited_as = 'A'
     and p.born >= 1970)
```

In fact I could turn the query into one with an uncorrelated subquery, executed only once, that returns all movies with at least one actor born in 1970 or later.

correlated



So to summarize we can switch correlated and uncorrelated subqueries, and turn them into joins.