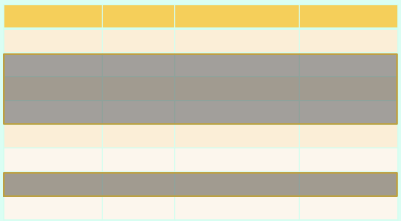
CS307

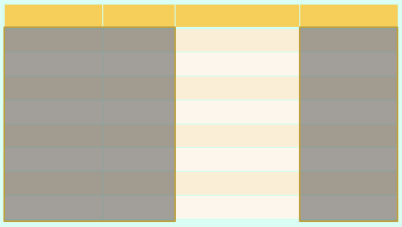
Database Principles

Stéphane Faroult
faroult@sustc.edu.cn

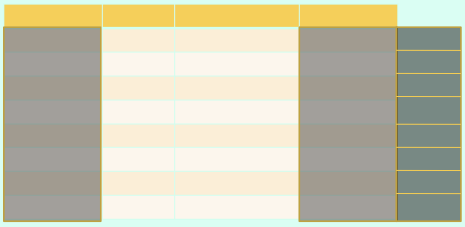
Liu Zijian liuzijian47@163.com



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

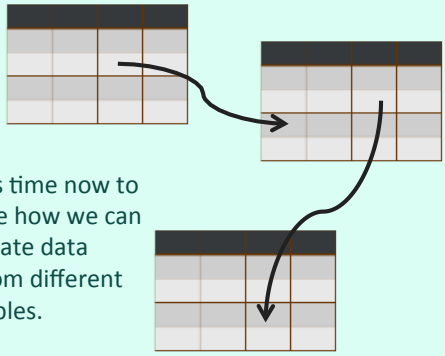


We have seen how we can only return some columns (called PROJECT by Codd), and that we must be careful not to return duplicates when we aren't returning a full key.



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 different tables.

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

Constraints

In practice, most joins will link tables through foreign key constraints. However, this is no obligation: relational joins are only driven by values. As long as columns contain the same values (even if it implies a datatype conversion), then you can link rows in different tables through these columns.

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	Ying hung boon sik	hk	1986
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
hk	Hong Kong	ASIA
it	Italy	EUROPE
ca	Canada	AMERICA
au	Australia	OCEANIA

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

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

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

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

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	Ying hung boon sik	hk 1986	hk Hong Kong	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 are building a kind of virtual super-wide (and badly normalized) table that unites columns from both tables.

countries

ru	Russia	EUROPE
us	United States	AMERICA
in	India	ASIA
gb	United Kingdom	EUROPE
fr	France	EUROPE
hk	Hong Kong	ASIA
it	Italy	EUROPE
ca	Canada	AMERICA
au	Australia	OCEANIA

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	Ying hung boon sik	hk 1986	hk Hong Kong	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 ...

```
 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.

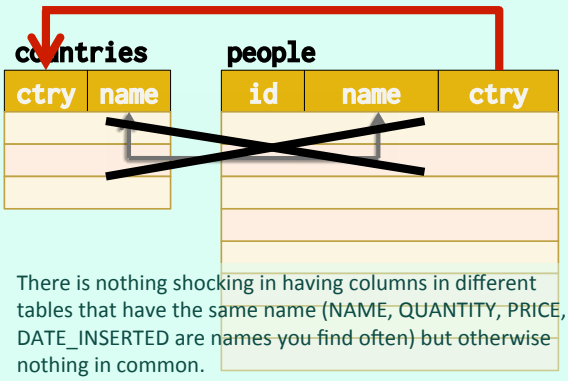
credits

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'
```

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.

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

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

As I am lazy and type badly, I usually even give a very 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 first_name, surname
from people p
     join credits  c
       on c.peopleid = p.peopleid
where 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.

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

Self-Join

A commonly used feature is what is called a "self-join, a join between a table and itself. The two different instances of the table in the query wear (metaphorically speaking) different caps.

21

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 c      -- child
     join people 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])x
    join ...
```

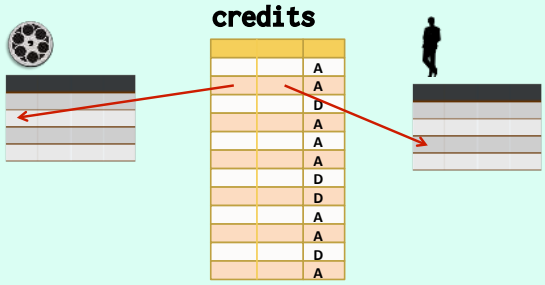
A join can as well be applied to a subquery seen as a virtual table, as long as the result of this subquery is a valid relation in Codd's sense. And if the result of a join is a valid relation, then we can join it again ...

```
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 surnames?

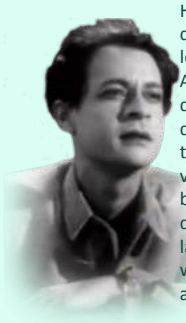
Let's write a relatively simple query. As you will see, even a simple query can let the door opened to problems.



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?



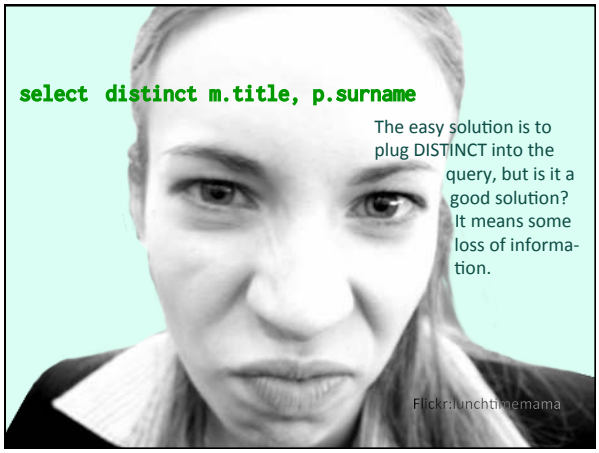
P.C. Barua

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.

দেবদাস 1935
देवदास 1936
দেবদাস 1937

surname ?

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



```
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

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'
```

Then you can join through the CREDITS table.

TRY IT!

Actors in any movie you like?

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

```
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'
```

Let's return to our British directors. 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'
```

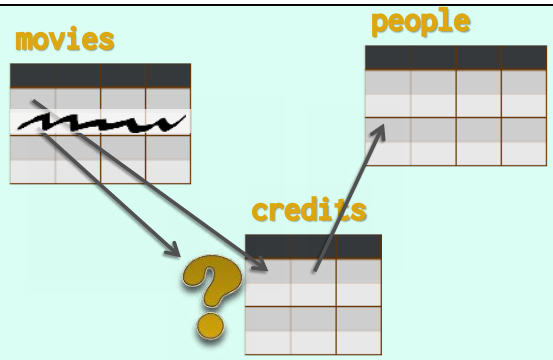
The newer syntax was designed to help differentiate between join conditions (after ON) and plain filtering conditions, and make more difficult to forget a join condition and get a Cartesian product, which is the combination of every row in a table with every row in another table.

```
select m.year_released, m.title,
       p.first_name, p.surname
from movies m
     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 = 2014
```

Now, if, instead of looking for the directors of British films you look for the directors of American ones in 2014, you'll discover something interesting ...

```
select m.year_released, m.title
from movies m
where m.country = 'us'
  and year_released = 2014
```

... which is that all films weren't listed in the previous query. We have for instance 'Captain America: The Winter Soldier' in the list returned by the query above, and not in the previous one.



The reason is that its directors, the Russo brothers, aren't credited for it in the database. No credits, no link with PEOPLE, the film vanishes.

```
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 m.year_released = 2014
```

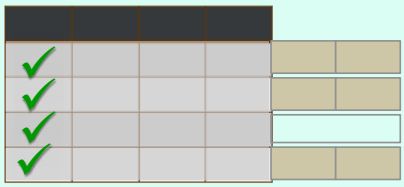
It's easier to understand why with the traditional SQL way of writing joins. Join conditions can also be interpreted as filtering conditions. If we can't find a row in CREDITS, then the condition isn't true.

If we want to see all 2014 American films 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)

outer join

inner

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

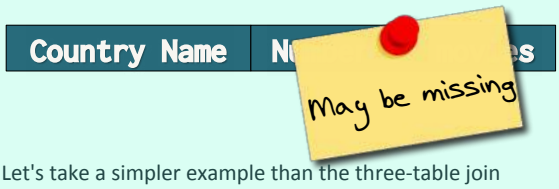


left outer join

~~right outer join~~

~~full outer join~~

Books always refer to three kind of outer joins. Only one is useful and I'll forget about anything but the LEFT OUTER JOIN. A right outer join can ALWAYS be rewritten as a left outer join. I have seen a full outer join used perhaps two or three times, the last time the query was significantly faster when rewritten without it.



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 films we have per country. We may have no films from smaller countries, or newer countries that haven't produced one film 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 films we have per country. This, of course, will only return countries for which there are films. If we use an inner join, they will be the only ones we'll see.

```
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.

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 films we'd rather see zero.
This is easy to do.

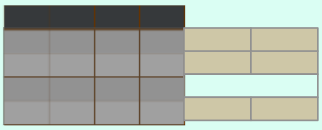
```
select c.country_name,
       case
         when x.number_of_movies is null then 0
         else x.number_of_movies
       end number_of_movies
from countries c
     left outer join
     (select country as country_code,
             count(*) as number_of_movies
      from movies
      group by country) x
          on x.country_code = c.country_code
```

We can use a CASE construct.

```
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 ❶
     left outer join table2 ❷
          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.

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 films, so MOVIES will be the leading table.

```
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.role = 'D'
  and m.country = 'gb'
```

Let's start with the regular, INNER JOIN query that will only show films for which the director is known. There are different ways to envision it.

```
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
           inner join credits c
              on c.movieid = m.movieid
            inner join people p
              on p.peopleid = c.peopleid) a
where a.credited_as = 'D'
  and a.country = 'gb'
```

We can say that we are building the list of credits for every film we have

then limit output to directors of British films.

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

Or we can say that we look for British films

and for directors

then return the names of people for which we find matching movieid values in both lists.

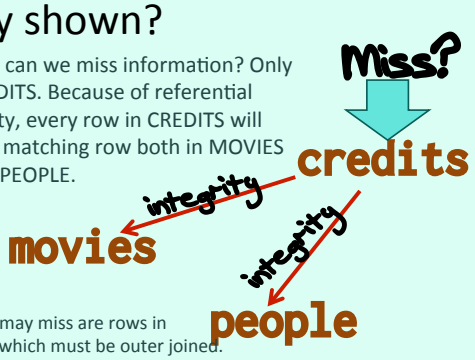
```
select a.year_released, a.title,
       a.first_name, a.surname
from (select m.year_released,
             m.title, m.country,
             p.first_name, p.surname,
             c.credited_as
      from movies m
           inner join credits c
              on c.movieid = m.movieid
            inner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

Basically, whether we filter first or last will have (spoiler alert!) some influence on performance, but not on returned data. We should ultimately see the same thing.

As you'll see, it's different with outer joins.

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.



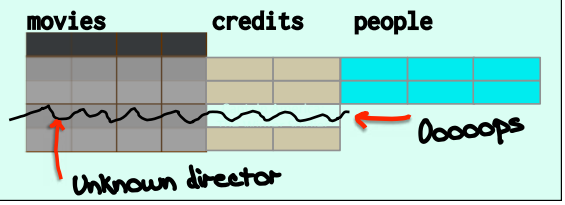
What we may miss are rows in CREDITS, which must be outer joined.

```
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.movied = m.movieid
    inner join people p          3
      on p.peopleid = c.peopleid
```

This condition will fail if we return a NULL row from 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'
  and a.country = 'gb'
```

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

outer join killer

But if the left outer join returns NULL, then CREDITED_AS cannot be 'D' and the film will disappear.

```
select a.year_released, a.title,
       a.first_name, a.surname
from (select m.year_released,
             m.title
             p.first_
             c.credit
      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'
```

Wrong!!

?

Testing for NULL might look a solution. Except that it's wrong.

Movies

People

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

movieid peopleid A

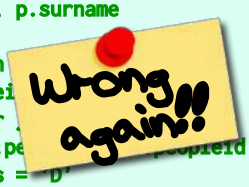
Credits

```
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'
```

A

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
          on c.moviei
          left outer
               on p.pe
where (c.credited_as =
       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 film.

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

I have told you that with an inner join I could as well first get British films, then directors. With outer joins this is how I MUST proceed if I want correct results.

```
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 films for MOVIES, but it is necessary to have a subquery that only returns directors.

Filter close to tables

In other words, with LEFT OUTER JOINs, apply all conditions before joining.

Joins → filtering

↓

qualifying

One thing is interesting with joins, which is that some joins participate in filtering, while others are only here to return additional information that make the result set more legible or intelligible.

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

Filtering

If I have a condition on the country NAME, then table COUNTRIES kind of drive the query.

```
select m.title, c.country_name
from movies m
  inner join countries c
     on c.country_code = m.country
where m.year_released = ...
```

Qualifying

If the conditions are only on MOVIES and if the only reason for joining on COUNTRIES is to return a name rather than a code, then the join is purely qualifying.

UNLESS

```
select m.title, c.country_name
from movies m
  inner join countries c
    on c.country_code = m.country
where m.date_released = ...
```

Significant?

unless finding a match is significant, because I might find only some values in the other table. With a foreign key constraint, it will never be the case: I'm sure to find the code in COUNTRIES.

```
select distinct
       m.title, c.country_name
from movies m
  inner join countries c
     on c.country_code = m.country
  inner join credits cr
    on cr.movieid = m.movieid
where m.year_released = ...
```

If I had a join with CREDITS, it would become significant: it would implicitly mean "only for films for which I know some of the people involved".

Outer joins → filtering

is null

↓

qualifying

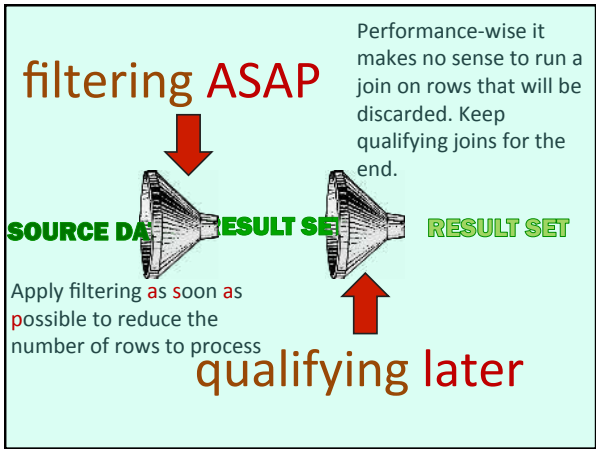
An outer join is always a qualifying join, unless it is associated with an IS NULL condition, meaning that not finding a match is significant (for instance: countries for which no film is found in MOVIES)

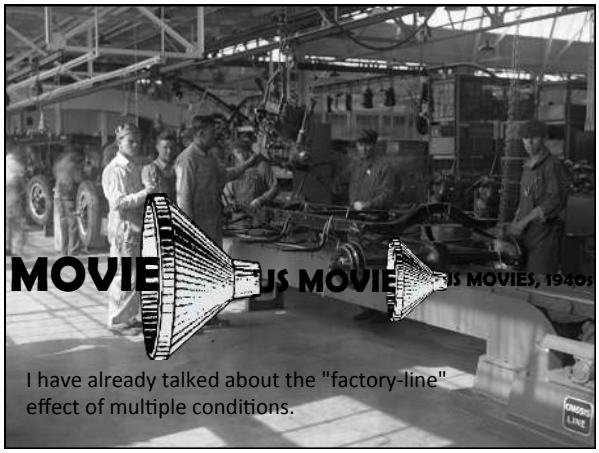
The real test is: what happens if I remove a join? Do I have more rows returned or the same number of rows as before?

~~Join~~ MORE rows?

YES → filtering

NO → qualifying






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

In particular, I have said that conditions linked by AND were like successive refining of result sets through successive queries.

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

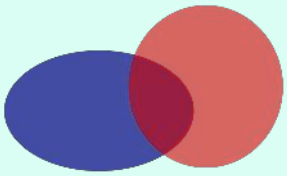
```
select *
from movies
where country_code = 'us'
  and year_released between 1940 and 1949
```

What I didn't say, though, is that this isn't true with OR. With OR, each new condition potentially adds MORE rows.

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

However, in the same way as successive queries can be substituted to AND, set operators can be substituted to OR conditions.

Set operators



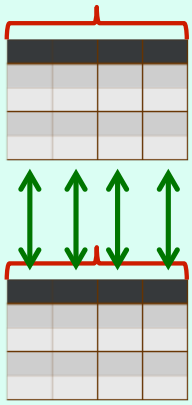
UNION

The most important (by far) set operator is UNION that takes two result sets and combine 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 films of the 1940s with British films of the 1940s (OR would be more efficient than two separate searches)



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

movies

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

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 films last year (for reference) and one with all the views for the current year, and you want to sum them both.

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

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

Goodfellas

2,356

Goodfellas

2,356

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 films, 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.

I first mentioned set operatorS with an S

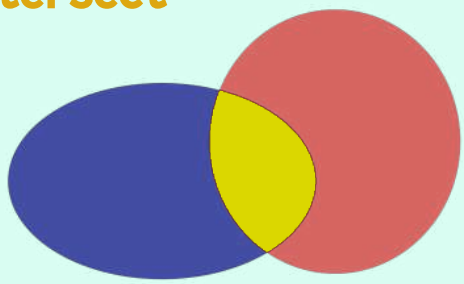
UNION is the most used set operator.

It's not the only one.

Let's see the others, and why they aren't as much used as UNION.

intersect

The first other set operator is 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, and rows present in one set and absent from another 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

Except of course that as we have a foreign key relationship between MOVIES and COUNTRIES and that we CANNOT have a code in MOVIES that isn't in COUNTRIES, all we need to do is collect the different country codes we find in MOVIES.

```
select distinct country
from movies
```

Important performance lesson: never do something that isn't required. The join and the intersect were doing work for nothing.

Different problem:

Countries for which we haven't any film.

Finding countries for which we have no film, 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
```

A set operator.

```
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, 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

Names of the countries in table **country** but not in table **movies**?

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

If we change the problem, we hardly need to change the LEFT OUTER JOIN query.

SET Operator:

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

If we want to keep the set operator, we need to wrap the query in a join with a table that already intervenes in the set operation ...

Ugly and inefficient.

To make several different tables intervene in a query, we have seen joins, and we have seen set operators. There is a third option, subqueries. You can use subqueries when what you return comes from a single table but depends on what you find in another table. Examples will make it clearer (hopefully).

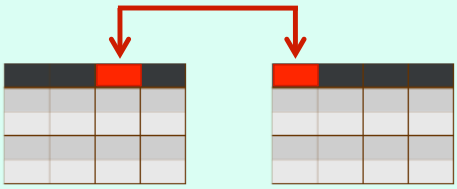
SUBQUERIES

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

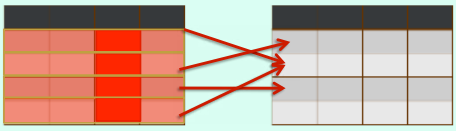
We have seen subqueries already: a query that is nested in a FROM clause is a subquery. But we can use subqueries anywhere, after the SELECT or in the WHERE clause.

correlation

But before I talk about subqueries, I must talk about something special: correlation.



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.

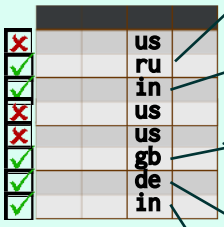
```
select m.title, m.year_released,
       c.country_name
from movies m
     join countries c
       on c.country_code = m.country
where m.country <> 'us'
```

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 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.




```
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

Both are common for subqueries in the WHERE clause.


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

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

I had mentioned IN () as 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 film 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 arelational 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 there is either a primary key or unique constraint) 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)