Principles of Database Systems (CS307)

Lecturer's Cut: SQL Practice

Yuxin Ma

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.

Introduction to SQL

How Do We Manage Data in a Database?

- Usually, a special language is needed
  - Be able to use a language to query a database
    - Either interactively or from within a program

- Query language
  - Query data
  - Modify data

Some History

- ALPHA
  - Codd's querying language

- Find the supplier names and locations of those suppliers who supply part 15.

$$(r_1[2],\ r_1[3]):\ P_1r_1 \wedge \exists P_2r_2(r_2[2] = 15 \wedge r_2[1] = r_1[1]).$$

- Find the locations of suppliers and the parts being supplied by them (omitting those suppliers who are supplying no parts at this time).

$$(r_1[3],\ r_2[2]):\ P_1r_1 \wedge P_2r_2 \wedge (r_1[1] = r_2[1]).$$

Codd, E.F., "Data Base Sublanguage Founded on the Relational Calculus", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access, and Control, San Diego.

Some History

- ALPHA
  - Codd's querying language

- Find the supplier names and locations of those suppliers who supply part 15.

$$(r_1[2],\ r_1[3]):\ P_1r_1 \wedge \exists P_2 r_2 (r_2[2] = 15 \wedge r_2[1] = r_1[1]).$$

- Find the locations of suppliers and the parts being supplied by them (omitting those suppliers who are supplying no parts at this time).

$$(r_1[3],\ r_2[2]):\ P_1r_1 \wedge P_2r_2 \wedge (r_1[1] = r_2[1]).$$

Codd, E.F., "Data Base Sublanguage Founded on the Relational Calculus", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access, and Control, San Diego.

However, it didn't excite enthusiasm at IBM

Some History

- "SEQUEL: A Structured English Query Language"
  - IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
  - An "easy" language, with an English-like syntax

Don Chamberlin with Ray Boyce (1974)

SEQUEL: A STRUCTURED ENGLISH QUERY LANGUAGE

by

Donald D. Chamberlin
Raymond F. Boyce

IBM Research Laboratory
San Jose, California

ABSTRACT: In this paper we present the data manipulation facility for a structured English query language (SEQUEL) which can be used for accessing data in an integrated relational data base. Without resorting to the concepts of bound variables and quantifiers SEQUEL identifies a set of simple operations on tabular structures, which can be shown to be of equivalent power to the first order predicate calculus. A SEQUEL user is presented with a consistent set of keyword English templates which reflect how people use tables to obtain information. Moreover, the SEQUEL user is able to compose these basic templates in a structured manner in order to form more complex queries. SEQUEL is intended as a data base sublanguage for both the professional programmer and the more infrequent data base user.

Computing Reviews Categories: 3.5, 3.7, 4.2

Key Words and Phrases: Query Languages
Data Base Management Systems
Information Retrieval
Data Manipulation Languages

Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A structured English query language. In Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control (SIGFIDET '74). Association for Computing Machinery, New York, NY, USA, 249–264. DOI:https://doi.org/10.1145/800296.811515

Some History

- SEQUEL was then renamed as Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features
  - Not all examples here may work on your particular system.

A huge problem that appears everywhere and many times

Year	Name	Alias	Comments
1986	SQL-86	SQL-87	First formalized by ANSI
1989	SQL-89	FIPS 127-1	Minor revision that added integrity constraints, adopted as FIPS 127-1
1992	SQL-92	SQL2, FIPS 127-2	Major revision (ISO 9075), *Entry Level* SQL-92 adopted as FIPS 127-2
1999	SQL:1999	SQL3	Added regular expression matching, recursive queries (e.g. transitive closure), triggers, support for procedural and control-of-flow statements, nonscalar types (arrays), and some object-oriented features (e.g. structured types), support for embedding SQL in Java (SQL/OLB) and vice versa (SQL/JRT)
2003	SQL:2003		Introduced XML-related features (SQL/XML), window functions, standardized sequences, and columns with autogenerated values (including identity columns)
2006	SQL:2006		ISO/IEC 9075-14:2006 defines ways that SQL can be used with XML. It defines ways of importing and storing XML data in an SQL database, manipulating it within the database, and publishing both XML and conventional SQL-data in XML form. In addition, it lets applications integrate queries into their SQL code with XQuery, the XML Query Language published by the World Wide Web Consortium (W3C), to concurrently access ordinary SQL-data and XML documents.[33]
2008	SQL:2008		Legalizes ORDER BY outside cursor definitions. Adds INSTEAD OF triggers, TRUNCATE statement,[34] FETCH clause
2011	SQL:2011		Adds temporal data (PERIOD FOR)[35] (more information at: Temporal database#History). Enhancements for window functions and FETCH clause.[36]
2016	SQL:2016		Adds row pattern matching, polymorphic table functions, JSON
2019	SQL:2019		Adds Part 15, multidimensional arrays (MDarray type and operators)

https://en.wikipedia.org/wiki/SQL

Standardization of SQL

Year	Name	Alias	Comments
1986	SQL-86	SQL-87	First formalized by ANSI
1989	SQL-89	FIPS 127-1	Minor revision that added integrity constraints, adopted as FIPS 127-1
1992	SQL-92	SQL2, FIPS 127-2	Major revision (ISO 9075), *Entry Level* SQL-92 adopted as FIPS 127-2
1999	SQL:1999	SQL3	Added regular expression matching, recursive queries (e.g. transitive closure), triggers, support for procedural and control-of-flow statements, nonscalar types (arrays), and some object-oriented features (e.g. structured types), support for embedding SQL in Java (SQL/OLB) and vice versa (SQL/JRT)
2003	SQL:2003		Introduced XML-related features (SQL/XML), window functions, standardized sequences, and columns with autogenerated values (including identity columns)
2006	SQL:2006		ISO/IEC 9075-14:2006 defines ways that SQL can be used with XML. It defines ways of importing and storing XML data in an SQL database, manipulating it within the database, and publishing both XML and conventional SQL-data in XML form. In addition, it lets applications integrate queries into their SQL code with XQuery, the XML Query Language published by the World Wide Web Consortium (W3C), to concurrently access ordinary SQL-data and XML documents.[33]
2008	SQL:2008		Legalizes ORDER BY outside cursor definitions. Adds INSTEAD OF triggers, TRUNCATE statement,[34] FETCH clause
2011	SQL:2011		Adds temporal data (PERIOD FOR)[35] (more information at: Temporal database#History). Enhancements for window functions and FETCH clause.[36]
2016	SQL:2016		Adds row pattern matching, polymorphic table functions, JSON
2019	SQL:2019		Adds Part 15, multidimensional arrays (MDarray type and operators)

https://en.wikipedia.org/wiki/SQL

Standardization of SQL (any other examples of standardization?)

Basic Syntax of SQL

```sql
select * from lab where time = '3-34';
```

`select ...`	• followed by the names of the columns you want to return
`from ...`	• followed by the name of the tables that you want to query
`where ...`	• followed by filtering conditions

Competing Languages of SQL

- QUEL, born at Berkeley, and associated with INGRES, was highly regarded
  - Ingres Database

QUEL ---------- Ingres → SEQUEL (SQL) ---------- Postgres

QUEL

```
create student(name = c10, age = i4, sex = c1, state = c2)

range of s is student
append to s (name = "philip", age = 17, sex = "m", state = "FL")

retrieve (s.all) where s.state = "FL"

replace s (age=s.age+1)

retrieve (s.all)

delete s where s.name="philip"
```

SQL

```
create table student(name char(10), age int, sex char(1), state char(2));

insert into student (name, age, sex, state) values ('philip', 17, 'm', 'FL');

select * from student where state = 'FL';

update student set age=age+1;

select * from student;

delete from student where name='philip';
```

Michael Stonebraker (1943-)
Turing Award 2014

Competing Languages of SQL

- QBE (Query by Example)
  - A visual querying tool (visual but with characters)
  - Created by IBM as well

```
id:
address:
time: 3-34
capacity:
teacher:
```

```
select * from lab where time = '3-34';
```

Moshé M. Zloof

Two Main Components for a Query Language

- From Codd's seminal paper:
  - A good database language should allow to deal as easily with contents (data) as containers (tables)
  - … Something that SQL does reasonably well

Two Main Components for a Query Language

- Data Definition Language (DDL)
  - The SQL data-definition language (DDL) allows the specification of information about relations, including:
    - The schema for each relation.
    - The type of values associated with each attribute.
    - The Integrity constraints
    - The set of indices to be maintained for each relation.
    - Security and authorization information for each relation.
    - The physical storage structure of each relation on disk.

Two Main Components for a Query Language

- Data Definition Language (DDL)
  - The SQL data-definition language (DDL) allows the specification of information about relations, including:
    - The schema for each relation
    - The type of values associated with each attribute
    - The Integrity constraints
    - The set of indices to be maintained for each relation
    - Security and authorization information for each relation
    - The physical storage structure of each relation on disk

```
create
alter
drop
```

Two Main Components for a Query Language

- Data Manipulation Language (DML)
  - Provides the ability to:
    - Query information from the database
    - Insert tuples into, delete tuples from, and modify tuples in the database.

```
select
insert
delete
update
```

Something about SQL

- Simple?
  - As you can see, it seems to be simple
  - But it becomes difficult when you combine operations
    - We will talk about it later
- Standard?
  - We have mentioned standardization of SQL before
  - However, no product fully implements it
    - Different product implements SQL differently
    - … and introduces dialects

SQL is one of a few languages where you spend more time thinking about how you are going to do things than actually coding them.

“Problems” in SQL

- SQL ≠ Relational Database
  - SQL wasn't designed as a "relationally correct" language
    - In some respects, it is very lax
    - But easy to use, however
    - And, easy to misuse
      - So, using it well is difficult
- Sometimes, you will get results even if the SQL is wrong
  - SQL is not as strict as C or Java; it will not give you error messages if your table cannot fit the requirement of the theory
    - “Wrong results” without warnings

“Problems” in SQL

- SQL ≠ Relational Database
  - SQL wasn't designed as a "relationally correct" language
    - In some respects, it is very lax
    - But easy to use, however
    - And, easy to misuse
      - So, using it well is difficult
- Sometimes, you will get results even if the SQL is wrong
  - SQL is not as strict as C or Java; it will not give you error messages if your table cannot fit the requirement of the theory
    - “Wrong results” without warnings

Be careful when designing your SQL queries

“Problems” in SQL

- Key propriety of relations (in Codd's original paper)

ALL ROWS ARE DISTINCT

  - This can be enforced for tables in SQL
    - (But you have to create your tables well)

  - But it is not enforced for query results in SQL
    - You must be extra-careful if the result of a query is the starting point for another query, which happens often. (i.e., combined queries)

Create Tables and Insert Data

Create Tables

- A comma-separated list of a column-name followed by spaces and a datatype specifies the columns in the table

Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

Example

```
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

Create Tables

- Table names
  - Case-insensitive (Usually, by default)
    - But, in some database systems, the case sensitivity is quite different
    - Try, or find the reference for the specific database system

```
CREATE TABLE tablename
CREATE TABLE TABLENAME
CREATE TABLE tABleNamE

/* Same table names */
```

For example, MariaDB is system-dependent with respect to case sensitivity

`https://mariadb.com/kb/en/identifier-case-sensitivity/`

Create Tables

- Table names
  - Case-insensitive (Usually, by default)
    - But, in some database systems, the case sensitivity is quite different
    - Try, or find the reference for the specific database system

For example, MariaDB is system-dependent with respect to case sensitivity

`https://mariadb.com/kb/en/identifier-case-sensitivity/`

```
CREATE TABLE tablename
CREATE TABLE TABLENAME
CREATE TABLE tABleNamE

/* Same table names */
```

Actually, keywords are case-insensitive as well

```
create table tablename
CREATE table TABLENAME
CReaTE tABle tABleNamE

/* Same keywords */
```

Create Tables

- Table names
  - Case-insensitive (Usually, by default)
    - But, in some database systems, the case sensitivity is quite different
    - Try, or find the reference for the specific database system
- Naming Convention
  - Underscores as word separators (instead of CamelCase in Java)

```
CREATE TABLE tablename
CREATE TABLE TABLENAME
CREATE TABLE tABleNamE

/* Same table names */
```

Actually, keywords are case-insensitive as well

```
create table tablename
CREATE table TABLENAME
CReaTE tABle tABleNamE

/* Same keywords */
```

Create Tables

- Table names
  - Case-insensitive (Usually, by default)
    - But, in some database systems, the case sensitivity is quite different
    - Try, or find the reference for the specific database system
- Naming Convention
  - Underscores as word separators (instead of CamelCase in Java)
- Be careful with double quotes
  - … which represents a “case-sensitive” name

```
CREATE TABLE tablename
CREATE TABLE TABLENAME
CREATE TABLE tABleNamE

/* Same table names */
```

Actually, keywords are case-insensitive as well

```
create table tablename
CREATE table TABLENAME
CReaTE tABle tABleNamE

/* Same keywords */
```

Create Tables

- An SQL relation is defined using the create table command:

```
create table r (
    A1 D1, A2 D2, ..., An Dn,
    (integrity-constraint1),
    ...,
    (integrity-constraintk)
)
```

  - `r` is the name of the relation
  - each `Ai` is an attribute name in the schema of relation `r`
  - `Di` is the data type of values in the domain of attribute `Ai`

Data Types

- Text data types
  - `char(length)` -- fixed-length strings
  - `varchar(max length)` -- non-fixed-length text
  - `varchar2(max length)` ORACLE -- Oracle's transformation of `varchar`
  - `clob` -- very long text (like GB-level text)
    - Or, `text` MySQL

Data Types

- Numerical types
  - `int` -- Integer (a finite subset of the integers that is machine-dependent)
  - `float(n)` -- Floating point number, with user-specified precision of at least *n* digits
  - `real` -- Floating point and double-precision floating point numbers, with machine-dependent precision
  - `numeric(p, d)`
    - Fixed point number, with user-specified precision of p digits, with d digits to the right of decimal point
    - E.g., `numeric(3,1)`, allows 44.5 to be stores exactly, but not 444.5 or 0.32
    - In SQL Server, it is also called `decimal`

Data Types

- Date types
  - `date` -- YYYY-MM-DD
  - `datetime` -- YYYY-MM-DD HH:mm:SS
  - `timestamp` -- YYYY-MM-DD HH:mm:SS
    - But it is in the UNIX timestamp
    - Value range: 1970-01-01 00:00:01 UTC - 2038-01-19 03:14:07 UTC
    - More reading about the “Year 2038 Problem” of the `timestamp` data type: `https://en.wikipedia.org/wiki/Year_2038_problem`

Data Types

- Binary data types
  - `raw(max length)`
  - `varbinary(max length)`
  - `blob` -- binary large object
  - `bytea` -- used in PostgreSQL

Constraints

- Can you find any problem in this statement?

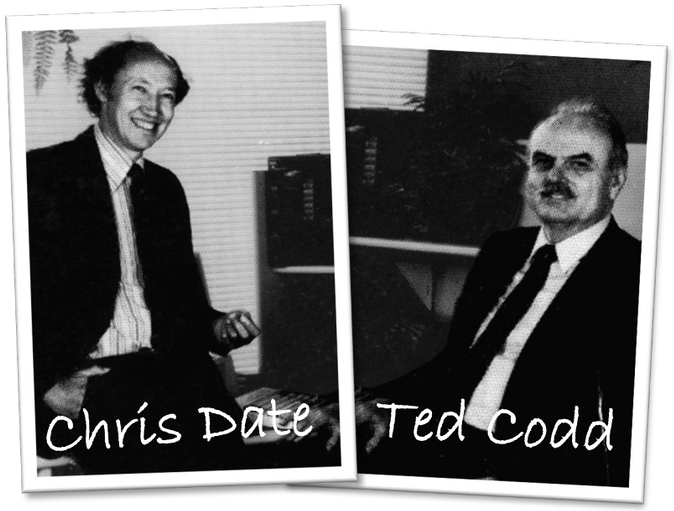
```
create table people (
    peopleid int,
    first_name varchar(30),
    surname varchar(30),
    born numeric(4),
    died numeric(4)
)
```

Constraints

- Can you find any problem in this statement?
  - It is valid and can be accepted by most DBMS
  - But it does nothing to enforce that we have a valid “relation” in Codd’s sense

```
create table people (
    peopleid int,
    first_name varchar(30),
    surname varchar(30),
    born numeric(4),
    died numeric(4)
)
```

Constraints



- Ted Codd and Chris Date
  - Worked on improving the relational theory
- Chris Date's work
  - Ensuring that only correct data that fits the theory can enter the database
  - Data inside the database remains correct
    - No need to double check for application programs

Constraints **are declarative rules that the DBMS will check every time new data will be added, when data is changed, or even when data is deleted, in order to prevent any inconsistency.**

* Any operation that violates a constraint fails and returns an error.

Constraints: Not NULL

- NULL values

```
create table people (
    peopleid int,
    first_name varchar(30),
    surname varchar(30),
    born numeric(4),
    died numeric(4)
)
```

- We don't want someone with no ID and name

```
create table people (
    peopleid int not null,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4),
    died numeric(4)
)
```

- Use `not null` to indicate that these columns are mandatory

Constraints: Not NULL

- NULL values

```
create table people (
    peopleid int,
    first_name varchar(30),
    surname varchar(30),
    born numeric(4),
    died numeric(4)
)
```

```
create table people (
    peopleid int not null,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4),
    died numeric(4)
)
```

- We don't want someone with no ID and name
- Use `not null` to indicate that these columns are mandatory

We can still have rows that with `NULL` values in the columns of `born`, `died`, and `first_name`

Constraints: Not NULL

- NULL values

```
create table people (
    peopleid int,
    first_name varchar(30),
    surname varchar(30),
    born numeric(4),
    died numeric(4)
)
```

```
create table people (
    peopleid int not null,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4),
    died numeric(4)
)
```

Why is only surname mandatory?

- It depends on the requirement. In this movie database case, some actors may be known as their stage names instead of real names. (Lady Gaga vs. Stefani Joanne Angelina Germanotta)

Takeaway: design your table according to the requirements

Constraints: Not NULL

- NULL values

```
create table people (
    peopleid int,
    first_name varchar(30),
    surname varchar(30),
    born numeric(4),
    died numeric(4)
)
```

```
create table people (
    peopleid int not null,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4),
    died numeric(4)
)
```

Similar to the column `born`

- We can either accept that
  - A row is created before we have information of that person's birth date
  - Or we require that the information should be found before entering the data

Comments

```
/* Multi-line
comments */

-- Single line comments, similar to double back-slashes in Java and C++

// *Some DBMS also support double back-slashes, like SQL Server
```

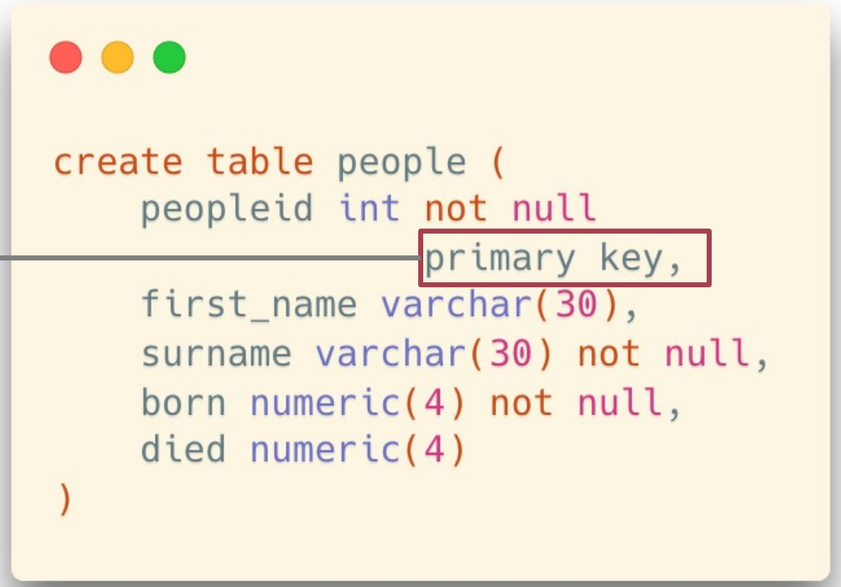
Comments

```
create table people (
    peopleid int not null,
    first_name varchar(30),
    surname varchar(30) not null,  -- the actual surname or the stage name
    born numeric(4) not null,  -- the birth date is mandatorybefore entering the data
    died numeric(4)
)
```

- Add comments to the definition of tables

Constraints: Primary Key

- The main key for the table, and indicates two things:
  - the value is mandatory
  - that the values are unique (no duplicates allowed in the column)



Constraints: Primary Key

- The main key for the table, and indicates two things:
  - the value is mandatory
  - that the values are unique (no duplicates allowed in the column)

```
create table people (
    peopleid int not null
                 primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4)
)
```

`primary key` implies `not null`, so `not null` here is redundant (but doesn't hurt)

Constraints: Unique

- So far, nothing would prevent us from entering two same rows with different IDs

peopleid	first_name	surname	born	died
1	Alfred	Hitchcock	1899	1980
2	Alfred	Hitchcock	1899	1980
3	....	...	...	...

```
create table people (
    peopleid int not null
                    primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4)
)
```

Constraints: Unique

- A unique constraint (on a combination of multiple columns)

peopleid	first_name	surname	born	died
1	Alfred	Hitchcock	1899	1980
2	Alfred	Hitchcock	1899	1980
3	….	…	…	…

```
create table people (
    peopleid int not null
                    primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4),
    unique (first_name, surname)
)
```

The combination of `(first_name, surname)` cannot be the same for any two rows

- But you still can have people with the same first name or surname, respectively

Constraints: Unique

- A unique constraint (on a single column)

peopleid	first_name	surname	born	died
1	Alfred	Hitchcock	1899	1980
2	Alfred	Hitchcock	1899	1980
3	....	...	...	...

No identical first names for any two people here

- But it is not what we want in this table

```
create table people (
    peopleid int not null
                    primary key,
    first_name varchar(30) unique,
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4)
)
```

Constraints: Check

- A column must satisfy a certain boolean expression test
  - The most generic constriant type

You must ensure that the person died after born

A useful trick to standardize names

- Such that there won't be rows with the same name of "Alfred Hitchcock", "ALFRED HITCHCOCK", and "alfred hitchcock".
- `upper(string)` is a function in PostgreSQL

```
create table people (
    peopleid int not null
                primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4),
    unique (first_name, surname),
    check (died - born >= 0),
    check (first_name = upper(first_name)),
    check (surname = upper(surname))
)
```

Named Constraints

- A name can be assigned to the constraints
  - … in order to refer to them easier in some other operations
    - PostgreSQL will give a name to the constraints if you don’t assign a name explicitly

```
create table people (
    peopleid int not null
                primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4),
    unique (first_name, surname),
    check (died - born >= 0),
    check (first_name = upper(first_name)),
    check (surname = upper(surname))
)
```

```
create table people (
    peopleid int not null
                primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4),
    unique (first_name, surname),
    constraint validate_birthdate check (died - born >= 0),
    constraint standardize_first_name check (first_name = upper(first_name)),
    constraint standardize_surname check (surname = upper(surname))
)
```

Referential Integrity

- Check constaints are static
  - Once it is written into the table, the criteria cannot be updated automatically

```
create table movies (
    movieid int not null primary key,
    title varchar (60) not null,
    country char (2) not null,
    year_released numeric(4) not null
    check (year_released >= 1895),
    unique (title, country, year_released)
)
```

- It is very difficult to perform static checks
  - Too many countries; country names and codes may change

Referential Integrity

- Check constaints are static
  - Once it is written into the table, the criteria cannot be updated automatically

```
create table movies (
    movieid int not null primary key,
    title varchar (60) not null,
    country char (2) not null,
    year_released numeric(4) not null
    check (year_released >= 1895),
    unique (title, country, year_released)
)
```

country_code	country_name	continent
US	United States	AMERICA
CN	China	ASIA
RU	Russia	EUROPE

Referential Integrity

- The `country` column in `movies` should be linked with the `country_code` column in another table (called reference table)

Foreign Key

- Format:
  - `foreign key (Am, ..., An ) references r`

```
create table movies (
    movieid int not null primary key,
    title varchar (60) not null,
    country char (2) not null,
    year_released numeric(4) not null
    check (year_released >= 1895),
    unique (title, country, year_released),
    foreign key (country) references country_list (country_code)
)
```

Meaning of this foreign key:

- The `country` column in this table (movies) refers to the `country_code` column in the table called `country_list`

Tip:

- `country_code` should be a key (primary key or unique) in the table `country_list`

Foreign Key

- Format:
  - `foreign key (Am, ..., An ) references r`

Movie ID	Movie Title	Country	Year
0	Citizen Kane	US	1941
1	La règle du jeu	FR	1939
2	North By Northwest	US	1959
3	Singin' in the Rain	US	1952
4	Rear Window	US	1954

Movie Entities

Directed By

Movie ID	Director ID
0	2
1	5
2	1

Foreign keys required in this table

Director ID	Director_ Firstname	Director_ Lastname	Born	Died
1	Alfred	Hitchcock	1899	1980
2	Orson	Welles	1915	1985
3	….	…	…	…

Director Entities

Foreign Key

- However, in some cases, foreign key can be a problem
  - Especially in big data processing applications
    - E.g., Alibaba Java Coding Guideline (阿里巴巴Java开发手册)

（三）SQL 语句

6．【强制】不得使用外键与级联，一切外键概念必须在应用层解决。

说明：以学生和成绩的关系为例，学生表中的 student_id 是主键，那么成绩表中的 student_id 则为外键。如果更新学生表中的 student_id，同时触发成绩表中的 student_id 更新，即为级联更新。外键与级联更新适用于单机低并发，不适合分布式、高并发集群；级联更新是强阻塞，存在数据库更新风暴的风险；外键影响数据库的插入速度。

https://ucc.alicdn.com/download/%E9%98%BF%E9%87%8C%E5%B7%B4%E5%B7%B4Java%E5%BC%80%E5%8F%91%E6%89%8B%E5%86%8C1.4.0.pdf

Summary: How to Create Tables

- Creating tables requires:
  - Proper modelling
  - Defining keys
  - Determining correct data types
  - Defining constraints

- Boring, but important
  - No further checks in the application programs; most things are ensured in the database layer

Updates to Tables

- Insert
  - `insert into instructor values ('10211', 'Smith', 'Biology', 66000);`
- Delete
  - Remove all tuples from the student relation
    - `delete from movies`

Updates to Tables

- Drop Table
  - `drop table r`
- Alter
  - `alter table r add A D`
    - where A is the name of the attribute (column) to be added to relation `r` and D is the data type of A.
    - All existing tuples in the relation are assigned null as the value for the new attribute.
  - `alter table r drop column A`
    - where A is the name of an attribute of relation r
    - Dropping of attributes not supported by many databases.
    - (There are some other things that can be “dropped”, including checks, foreign keys, indexes, etc.)

Updates to Tables

- More about insert

```
create table lab (
    id serial primary key,
    address varchar(20) not null,
    time varchar(20) not null,
    capacity int,
    teacher varchar(20),
    unique (address,time)
);
```

Values must match column names one by one

```
insert into lab (address, time, capacity, teacher) values ('402','2-78',36,'yueming');

insert into lab (address, time, teacher) values ('402','2-78','yueming');

insert into lab (address, time, teacher) values ('408','2-78','o''relly');
```

Updates to Tables

- More about insert

```
create table lab (
    id serial primary key,
    address varchar(20) not null,
    time varchar(20) not null,
    capacity int,
    teacher varchar(20),
    unique (address,time)
);


insert into lab (address, time, capacity, teacher) values ('402','2-78',36,'yueming');

insert into lab (address, time, teacher) values ('402','2-78','yueming');

insert into lab (address, time, teacher) values ('408','2-78','o''relly');
```

Missing columns and values for "nullable" columns are allowed

- … and a NULL willl be inserted

* But if you miss a mandatory column (such as address), an error will occur.

Updates to Tables

- More about insert

```
create table lab (
    id serial primary key,
    address varchar(20) not null,
    time varchar(20) not null,
    capacity int,
    teacher varchar(20),
    unique (address,time)
);


insert into lab (address, time, capacity, teacher) values ('402','2-78',36,'yueming');

insert into lab (address, time, teacher) values ('402','2-78','yueming');

insert into lab (address, time, teacher) values ('408','2-78','o''relly');
```

* Use two single quotes to represent a single quote in the content (i.e., escape charactor)

Basics for Selecting Data from One Table

Select

- select * from tablename
  - The `select` clause lists the attributes desired in the result of a query
  - To display the full content of a table, you can use select *
    - * : all columns

```
select A1, A2, ..., An
 from r1, r2, ..., rm
       where P
```

Select

- `select * from` [tablename]
  - The `select` clause lists the attributes desired in the result of a query
  - To display the full content of a table, you can use select *
    - * : all columns

```
select A1, A2, ..., An
 from r1, r2, ..., rm
      where P
```

- Such a query is frequently used in interactive tools (especially when you don't remember column names ...)
  - But you should not use it, though, in application programs

Restrictions

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

peopleid	first_name	surname	born	died

Restrictions

- Filtering
  - Performed in the “`where`” clause
  - Conditions are usually expressed by a column name
    - … followed by a comparison operator and the value to which the content of the column is compared
  - Only rows for which the condition is true will be returned

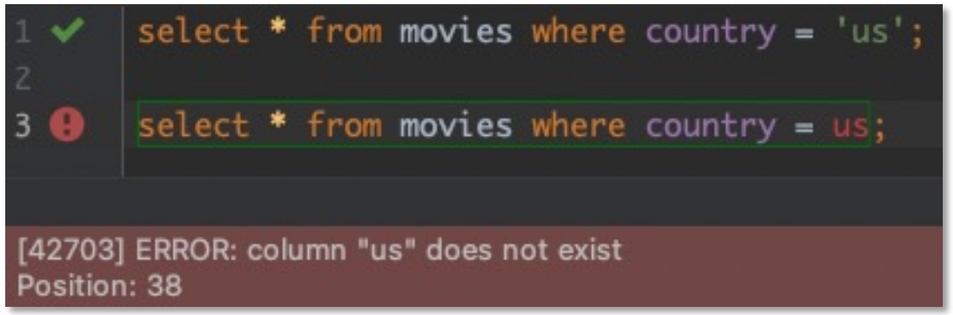
```
select * from movies where country = 'us';
```

Comparison

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

String Constants

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



Filtering

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

```
select country from movies;
```

	country
1	ru
2	eg
3	ma
4	ar
5	in
6	in
7	pk
8	dk
9	jp
10	eg
11	us
12	ca
13	ru
14	be
15	br
16	my
17	cn
18	de

Select without From or Where

- An attribute can be a literal with no from clause

```
select '437'
```

  - Results is a table with one column and a single row with value “437”
  - Can give the column a name using:

```
select '437' as FOO
```

- An attribute can be a literal with from clause

```
select 'A’ from movies
```

  - Result is a table with one column and N rows (number of tuples in the movies table), each row with value “A”

Select without From or Where

- An attribute can be a literal with no from clause

```
select '437'
```

A common way to test expressions

  - Results is a table with one column and a single row with value “437”
  - Can give the column a name using:

```
select '437' as FOO
```

- An attribute can be a literal with from clause

```
select 'A’ from movies
```

  - Result is a table with one column and N rows (number of tuples in the movies table), each row with value “A”

Arithmetic Expression

- The `select` clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples

	runtime
1	161
2	102
3	90
4	94
5	130
6	159
7	<null>
8	102
9	108
10	<null>
11	106
12	<null>
13	100
14	95
15	<null>

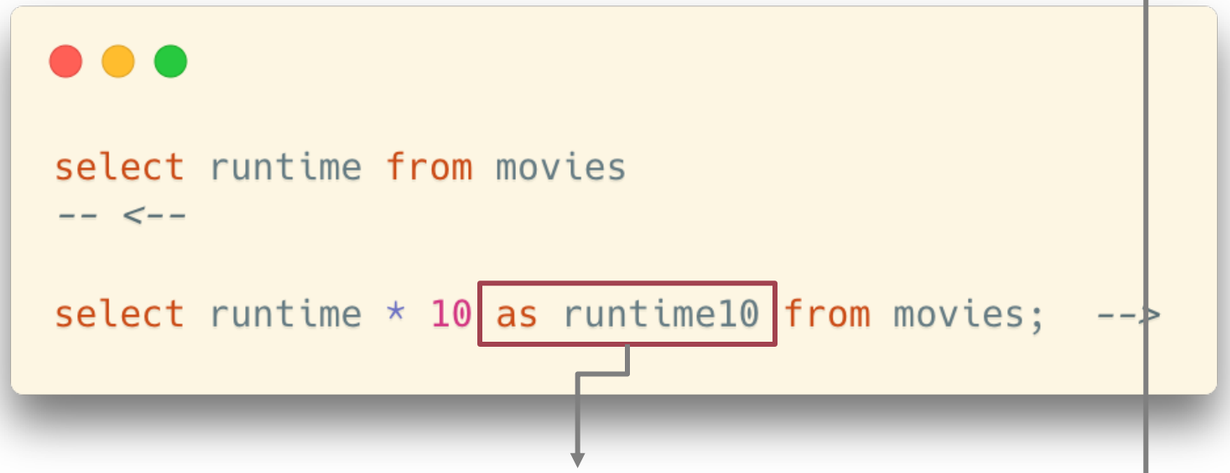
```
select runtime from movies
-- <--

select runtime * 10 as runtime10 from movies;  -->
```

	runtime10
1	1610
2	1020
3	900
4	940
5	1300
6	1590
7	<null>
8	1020
9	1080
10	<null>
11	1060
12	<null>
13	1000
14	950
15	<null>

Arithmetic Expression

- The `select` clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples

	runtime
1	161
2	102
3	90
4	94
5	130
6	159
7	<null>
8	102
9	108
10	<null>
11	106
12	<null>
13	100
14	95
15	<null>



	runtime10
1	1610
2	1020
3	900
4	940
5	1300
6	1590
7	<null>
8	1020
9	1080
10	<null>
11	1060
12	<null>
13	1000
14	950
15	<null>

Logical Connectives

- and, or, not
  - Just like in programming languages
  - All logical operators have different precedence
    - For example, and is "stronger" than or.

Table 1-1. Operator Precedence (decreasing)

Operator/Element	Associativity	Description
::	left	PostgreSQL-style typecast
[ ]	left	array element selection
.	left	table/column name separator
-	right	unary minus
^	left	exponentiation
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
IS		test for TRUE, FALSE, UNKNOWN, NULL
ISNULL		test for NULL
NOTNULL		test for NOT NULL
(any other)	left	all other native and user-defined operators
IN		set membership
BETWEEN		containment
OVERLAPS		time interval overlap
LIKE ILIKE		string pattern matching
< >		less than, greater than
=	right	equality, assignment
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

https://www.postgresql.org/docs/7.2/sql-precedence.html

Logical Connectives

- and, or, not
  - Just like in programming languages
  - All logical operators have different precedence
    - For example, and is "stronger" than or.

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

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

Differences?

Logical Connectives

- Use parentheses to specify that the or should be evaluated before the and, and that the conditions filter
  - 1) British or American films
  - 2) That were released in the 1940s

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


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

Logical Connectives

- Question:
  - Find the Chinese movies from the 1940s and American movies from the 1950s

Logical Connectives

- Question:
  - Find the Chinese movies from the 1940s and American movies from the 1950s

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

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

- The parentheses make the statement easier to understand

Logical Connectives

- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.
  - Note that there are two ways to write “not equal to”: != and <>
  - Comparisons can be applied to results of arithmetic expressions
- Beware that "bigger" and "smaller" have a meaning that depends on the data type
  - It can be tricky because most products implicitly convert one of the sides in a comparison between values of differing types

```
2 < 10       -- true
'2' < '10'   -- false

'2-JUN-1883'>'1-DEC-2056'  -- single-quoted, treated as strings but not dates
```

Logical Connectives

- `in()`
  - It can be used as the equivalent for a series of equalities with `OR` (it has also other interesting uses)
  - It may make a comparison clearer than a parenthesized expression

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

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

Logical Connectives

- Negation
  - All comparisons can be negated with `NOT`

```
-- exclude all movies selected in the previous page

where not ((country in ('us', 'gb')) and (year_released between 1940 and 1949))
where (country not in ('us', 'gb')) or (year_released not between 1940 and 1949)  -- equivalent query
```

`between` Comparison Operator

- `between ... and ...`
  - shorthand for: >= and <=

```
year_released between 1940 and 1949

-- It's shorthand for this:
year_released >= 1940 and year_released <= 1949
```

`between` Comparison Operator

- `between ... and ...`
  - shorthand for: >= and <=

```
year_released between 1940 and 1949

-- It's shorthand for this:
year_released >= 1940 and year_released <= 1949
```

not “<”

like

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

like

```sql
select * from movies where title not like '%A%' and title not like '%a%';

select * from movies where upper(title) not like '%A%';
-- not recommended due to the performance cost of upper()
```

- This expression for instance returns films the title of which doesn't contain any A
  - This A might be the first or last character as well
  - Note that if the DBMS is case sensitive, you need to cater both for upper and lower case
  - Function calls could slow down queries; use with caution

Date

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

```
DD/MM/YYYY
MM/DD/YYYY
YYYY/MM/DD
```

Date

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

- Whenever you are comparing data of slightly different types, you should use functions that "cast" data types
  - It will avoid bad surprises
  - The functions don't always bear the same names but exist with all products
- Default formats vary by product, and can often be changed at the DBMS level
  - So, better to use explicit date types and functions other than strings
  - Conversely, you can format something that is internally stored as a date and turn it into a character string that has almost any format you want

Date and Datetime

- If you compare datetime values to a date (without any time component) the SQL engine will not understand that the date part of the datetime should be equal to that date
  - Rather, it will consider that the date that you have supplied is actually a datetime, with the time component that you can read below
    - `date('2020-03-20')` is equal to `datetime('2020-03-20 00:00:00')`
- Date functions
  - Many useful date functions when manipulating date and datetime values
  - However, most of them are DBMS-dependent

```sql
select date_eq_timestamp(date('2018-03-12'), date('2018-02-12') + interval '1 month');  -- true
```

NULL

- In a language such as Java, you can compare a reference to `null`, because `null` is defined as the ‘0’ address.
  - In C, you can also compare a pointer to `NULL` (pointer is C-speak for reference)

NULL

- Not in SQL, where `NULL` denotes that a value is missing
  - `Null` in SQL is not a value
    - … and if it's not a value, hard to say if a condition is true.
    - A lot of people talk about "null values", but they have it wrong
  - Most expression with `NULL` is evaluated to `NULL`

```sql
select * from movies where runtime is null;

select * from movies where runtime = null;  -- warning in DataGrip; not the same as "is null"
```

Some Functions

- Show DDL of a table

```
desc movies;  -- Oracle, MySQL

describe table movies  -- IBM DB2

\d movies  -- PostgreSQL

.schema movies  -- SQLite
```

Some Functions – Compute and Derive

- One important feature of SQL is that you don’t need to return data exactly as it was stored
  - Operators, and many (*mostly DBMS specific*) functions allow to return transformed data

Some Functions

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

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

	movie_release
1	Variety was released in 2008
2	Inglourious Basterds was released in 2009
3	La grande vadrouille was released in 1966
4	Pulp Fiction was released in 1994
5	Schindler's List was released in 1993
6	Star Wars was released in 1977
7	The Dark Knight was released in 2008
8	The Godfather was released in 1972
9	The Shawshank Redemption was released in 1994
10	Titanic was released in 1997
11	Charade was released in 1963
12	North by Northwest was released in 1959
13	Singin' in the Rain was released in 1952
14	Rear Window was released in 1954
15	City Lights was released in 1931

Some Functions

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

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

	movie_release
1	Variety was released in 2008
2	Inglourious Basterds was released in 2009
3	La grande vadrouille was released in 1966
4	Pulp Fiction was released in 1994
5	Schindler's List was released in 1993
6	Star Wars was released in 1977
7	The Dark Knight was released in 2008
8	The Godfather was released in 1972
9	The Shawshank Redemption was released in 1994
10	Titanic was released in 1997
11	Charade was released in 1963
12	North by Northwest was released in 1959
13	Singin' in the Rain was released in 1952
14	Rear Window was released in 1954
15	City Lights was released in 1931

Note that you can give a name to an expression

- This will be used as column header
- It also becomes a "virtual column" if you turn the query into a "virtual table"

Some Functions

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

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

Although YEAR_RELEASED is actually a number, it's implicitly turned into a string by the DBMS.

- In that case it's not a big issue, but it would be better to use a function to convert explicitly.

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

Some Functions

- When to use functions
  - An example of showing a result that isn't stored as such is computing an age
    - You should never store an age; it changes all the time!
    - If you want to display the age of people who are alive, you must compute their age by subtracting the year when they were born from the current year.

Some Functions

- When to use functions
  - An example of showing a result that isn't stored as such is computing an age
    - You should never store an age; it changes all the time!
    - If you want to display the age of people who are alive, you must compute their age by subtracting the year when they were born from the current year.

  - In the table people:
    - Alive – died is `null`
    - Age: `<this year> - born`

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

7	7	Caroline	Aaron	1952	<null>	F
8	8	Quinton	Aaron	1984	<null>	M
9	9	Dodo	Abashidze	1924	1990	M

Some Functions

- Numerical functions

```
round(3.141592, 3)  -- 3.142
trunc(3.141592, 3)  -- 3.141
```

- More string functions

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

Some Functions

- Type casting
  - `cast(column as type)`

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

Case

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

```
CASE input_expression
    WHEN when_expression THEN result_expression
    [ ...n ]
    [ELSE else_result_expression]
END
```

```
CASE
    WHEN Boolean_expression THEN result_expression
    [ ...n ]
    [ELSE else_result_expression]
END
```

Case

- Example 1: Show the corresponding words of the gender abbreviations

```
select peopleid, surname,
    case gender
        when 'M' then 'male'
        when 'F' then 'female'
    end as gender_2
from people
where died is null;
```

*Similar to the switch-case statement in Java and C

Case

- Example 2: Decide whether someone’s age is older/younger than a pivot

Case

- Example 2: Decide whether someone's age is older/younger than a pivot

A horrible solution!

```
case age
    when 30 then 'younger than 44'
    when 31 then 'younger than 44'
    when 32 then 'younger than 44'
    when 33 then 'younger than 44'
    when 34 then 'younger than 44'
    when 35 then 'younger than 44'
    when 36 then 'younger than 44'
    ...
    when 43 then 'younger than 44'
    when 44 then '44 years old'
    when 45 then 'older than 44'
    ...
end as status
```

Case

- Example 2: Decide whether someone's age is older/younger than a pivot
  - CASE

```
select peopleid, surname,
    case (date_part('year', now()) - born > 44)
        when true then 'older than 44'
        when false then 'younger than 44'
        else '44 years old'
    end as status
from people
where died is null;
```

Case

- Example 2: Decide whether someone's age is older/younger than a pivot
  - CASE
  - CASE WHEN

```
select peopleid, surname,
    case
        when (date_part('year', now()) - born > 44) then 'older than 44'
        when (date_part('year', now()) - born < 44) then 'younger than 44'
        else '44 years old'
    end as status
from people
where died is null;
```

Case

- Example 2: Decide whether someone's age is older/younger than a pivot
  - CASE
  - CASE WHEN

```
select peopleid, surname,
    case
        when (date_part('year', now()) - born > 44) then 'older than 44'
        when (date_part('year', now()) - born < 44) then 'younger than 44'
        else '44 years old'
    end as status
from people
where died is null;
```

The ELSE branch

- Return a default value when all `when` criteria are not met
- If no else, `NULL` will be returned

Case

- About the `NULL` value
  - Use the "`is null`" criteria

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

More on Retrieving Data

Distinct

Distinct

- No duplicated identifier
  - **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, i.e., no duplicates

Distinct

- If we run a query such as the one below
  - Many identical rows
    - In other words, we may be obtaining a table, but it's not a relation because many rows cannot be distinguished

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

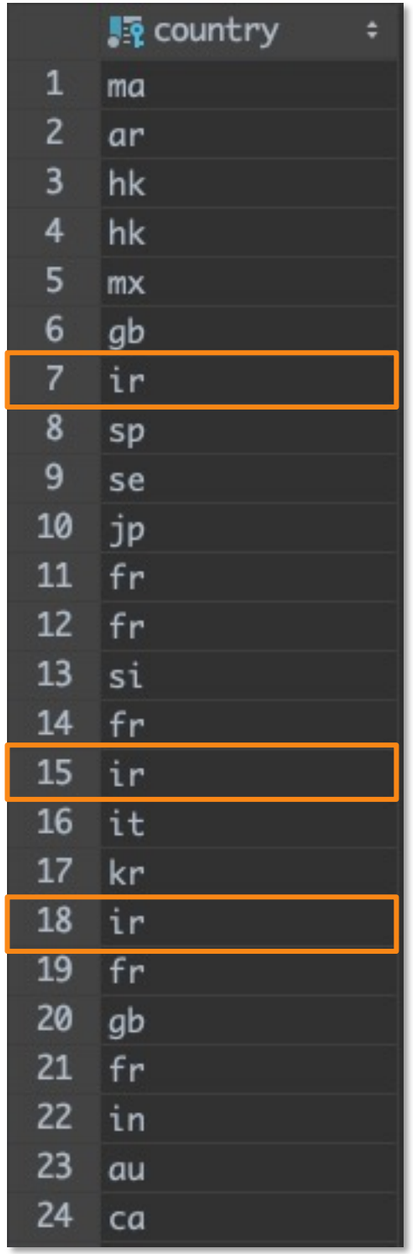
	country
1	ma
2	ar
3	hk
4	hk
5	mx
6	gb
7	ir
8	sp
9	se
10	jp
11	fr
12	fr
13	si
14	fr
15	ir
16	it
17	kr
18	ir
19	fr
20	gb
21	fr
22	in
23	au
24	ca

Duplicated country codes in the query result

- But their original rows are not considered duplicated tuples

Distinct

- The result of the query is in fact completely uninteresting
  - Whenever we are only interested in countries in table movies, it can only be for one of two reasons:
    - See a list of countries that have movies
    - Or, for instance, see which countries appear most often



	country
1	ma
2	ar
3	hk
4	hk
5	mx
6	gb
7	ir
8	sp
9	se
10	jp
11	fr
12	fr
13	si
14	fr
15	ir
16	it
17	kr
18	ir
19	fr
20	gb
21	fr
22	in
23	au
24	ca

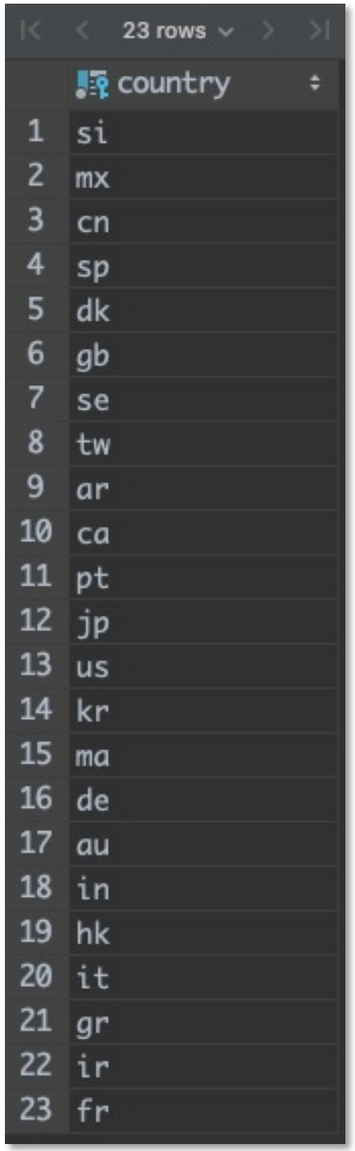
Duplicated country codes in the query result

- But their original rows are not considered duplicated tuples

Distinct

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

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



No duplicated results in the country code list now

- All of them are different now, and hence it is a relation!

Distinct

- Multiple columns after the keyword `distinct`
  - It will eliminate those rows where all the selected fields are identical
  - The selected combination `(country, year_released)` will be identical

```
select distinct country, year_released
from movies
where year_released in (2000,2001);
```

44 rows

	country	year_released
1	nz	2001
2	ar	2001
3	mx	2000
4	kr	2001
5	in	2001
6	ma	2000
7	si	2000
8	ca	2001
9	uy	2001
10	pt	2001
11	fr	2000
12	de	2000
13	us	2001
14	au	2001
15	au	2000
16	hu	2001
17	ie	2001
18	sp	2000
19	in	2000
20	us	2000
21	nl	2001
22	hk	2001
23	tw	2000

More on Retrieving Data

Aggregate Functions

Aggregate Functions

- Statistical functions
  - When we are interested in what we might call countrywide characteristics, such as how many movies released, we use Aggregate Functions.
  - 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

Aggregate Functions

- To compute an aggregated result, we'll first retrieve data
  - Here, all rows are in the table

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

country	year_released	title
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
in	1975	Sholay
in	1955	Pather Panchali
jp	1954	Shichinin no Samurai

Note: Just for demonstration purpose, not the real data in the table `movie`

Aggregate Functions

- To compute an aggregated result, we'll first retrieve data
  - Here, all rows are in the table
- Then, data will be regrouped according to the value in one or several columns

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

Grouped according to country

- Rows with the same value will be grouped together

country	year_released	title
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
in	1975	Sholay
in	1955	Pather Panchali
jp	1954	Shichinin no Samurai

Note: Just for demonstration purpose, not the real data in the table `movie`

Aggregate Functions

- We say that we want to "group by country"
  - … and, for each country, the aggregate function `count(*)` says how many movies we have
    - "how many movies" = "how many rows"

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

- The query result
  - One row for each group
  - The statistical value is attached in another column

	country	number_of_movies
1	fr	571
2	ke	1
3	si	1
4	eg	11
5	nz	23
6	bg	4
7	ru	153
8	gh	1
9	pe	4
10	hr	1
11	sg	5
12	mx	59
13	cn	200

Aggregate Functions

By the way, we can **rename** the column of the aggregate function, like below

- We say that we want to "group by country"
  - … and, for each country, the aggregate function `count(*)` says how many movies we have
    - "how many movies" = "how many rows"

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

	country	number_of_movies
1	fr	571
2	ke	1
3	si	1
4	eg	11
5	nz	23
6	bg	4
7	ru	153
8	gh	1
9	pe	4
10	hr	1
11	sg	5
12	mx	59
13	cn	200

- The query result
  - One row for each group
  - The statistical value is attached in another column

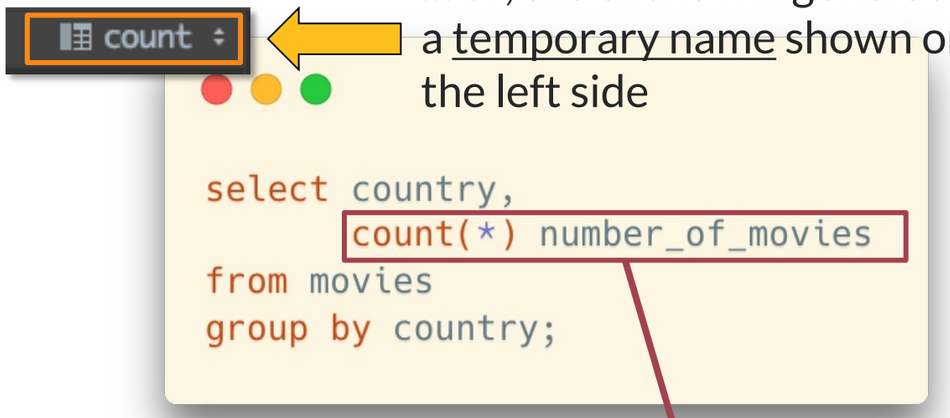
Aggregate Functions

- We say that we want to "group by country"
  - … and, for each country, the aggregate function `count(*)` says how many movies we have
    - "how many movies" = "how many rows"

- The query result
  - One row for each group
  - The statistical value is attached in another column

By the way, we can **rename** the column of the aggregate function, like below

- … or, the client will generate a temporary name shown on the left side



	country	number_of_movies
1	fr	571
2	ke	1
3	si	1
4	eg	11
5	nz	23
6	bg	4
7	ru	153
8	gh	1
9	pe	4
10	hr	1
11	sg	5
12	mx	59
13	cn	200

Aggregate Functions

- We say that we want to "group by country"
  - … and, for each country, the aggregate function `count(*)` says how many movies we have
    - "how many movies" = "how many rows"

Caution: The table `movie` must be a relation (no duplicated movie records)

- … or, the counting result will not reflect the actual number of movies

- The query result
  - One row for each group
  - The statistical value is attached in another column

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

	country	number_of_movies
1	fr	571
2	ke	1
3	si	1
4	eg	11
5	nz	23
6	bg	4
7	ru	153
8	gh	1
9	pe	4
10	hr	1
11	sg	5
12	mx	59
13	cn	200

Aggregate Functions

- Group on several columns
  - Every column that isn't an aggregate function and appears after `select` must also appear after `group by`

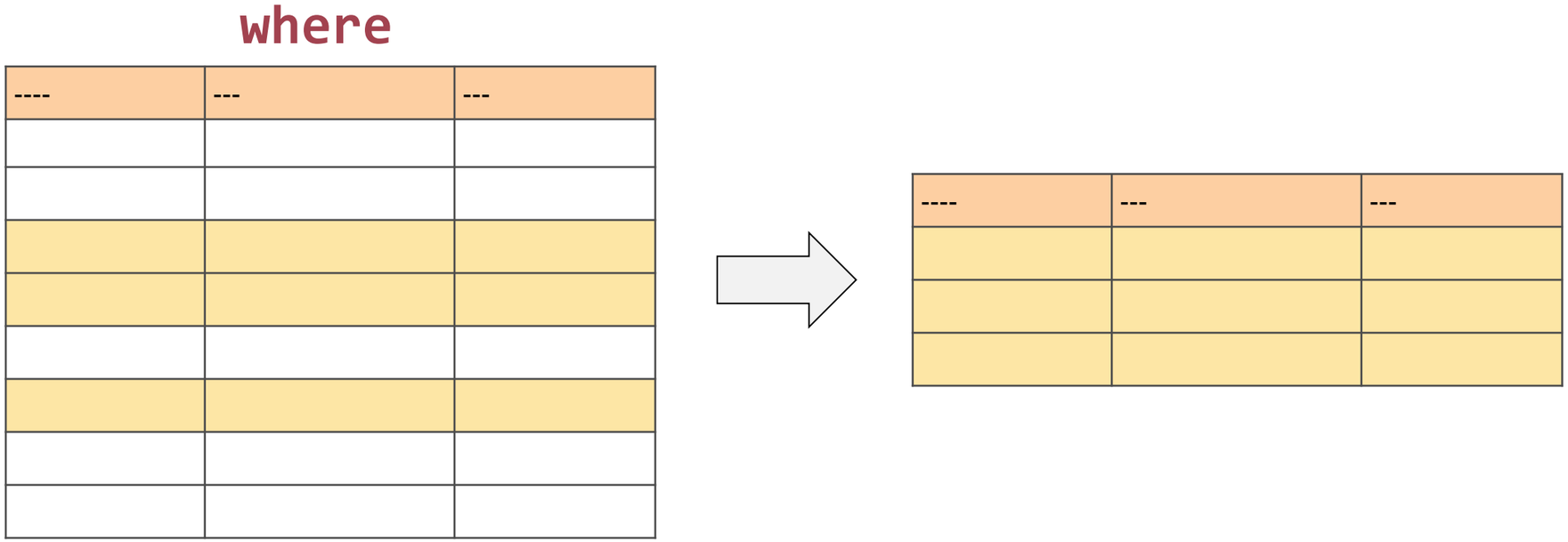
```
select country,
       year_released,
       count(*) number_of_movies
from movies
group by country, year_released
```

The combination of the countries and released years will appear in the result

	country	year_released	number_of_movies
1	us	1939	46
2	cn	2016	13
3	nl	2008	1
4	it	1960	10
5	ch	2011	1
6	us	1931	33
7	fr	1961	11
8	cn	2007	5
9	mn	2007	1
10	nz	2010	1
11	de	1974	2
12	au	1978	4
13	us	1935	36
14	eg	1987	1

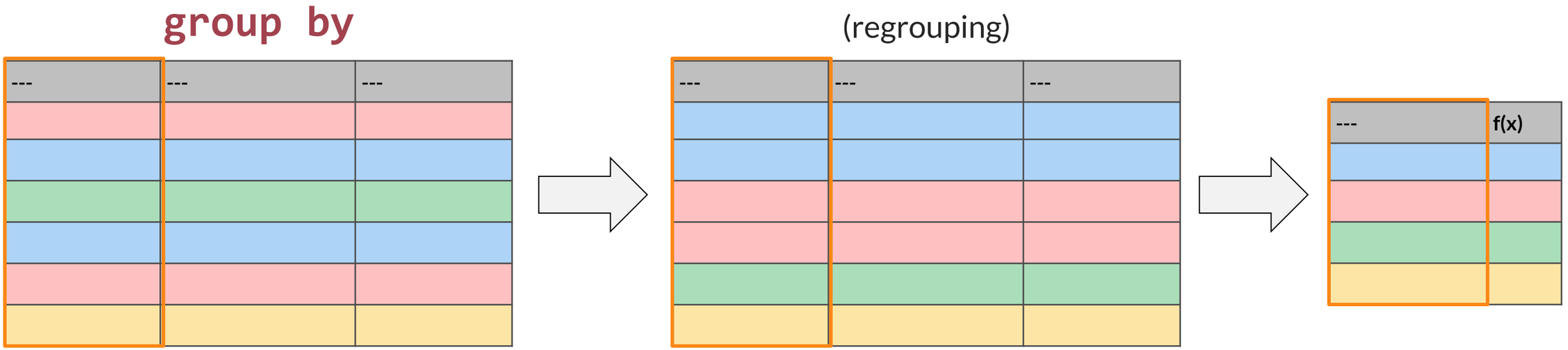
Aggregate Functions

- Beware of some performance implications
  - When you apply a simple `where` filter, you can start returning rows as soon as you have found a match.



Aggregate Functions

- Beware of some performance implications
  - 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.



Aggregate Functions

`count(*)/count(col), min(col), max(col), stddev(col), avg(col)`

- These aggregate function examples exist in almost all products
  - Most products implement other functions
  - Some work with any datatype, others only work with numerical columns

  - It is strongly recommended to refer to the database manual for details
    - For example, SQLite doesn't have `stddev()` which computes the standard deviation

Aggregate Functions

- *Earliest release year by country?*

Aggregate Functions

- *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
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

Aggregate Functions

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

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

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)

Aggregate Functions

- 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).

Aggregate Functions

SORT: Time complexity of sorting algorithms: O(n*log(n))

- 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.

Aggregate Functions

- 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
  - ... In all, optimizers don't all behave the same.

Aggregate Functions

- *Nulls?*

- 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?

- known + unknown = unknown

Aggregate Functions

- ***Nulls?***

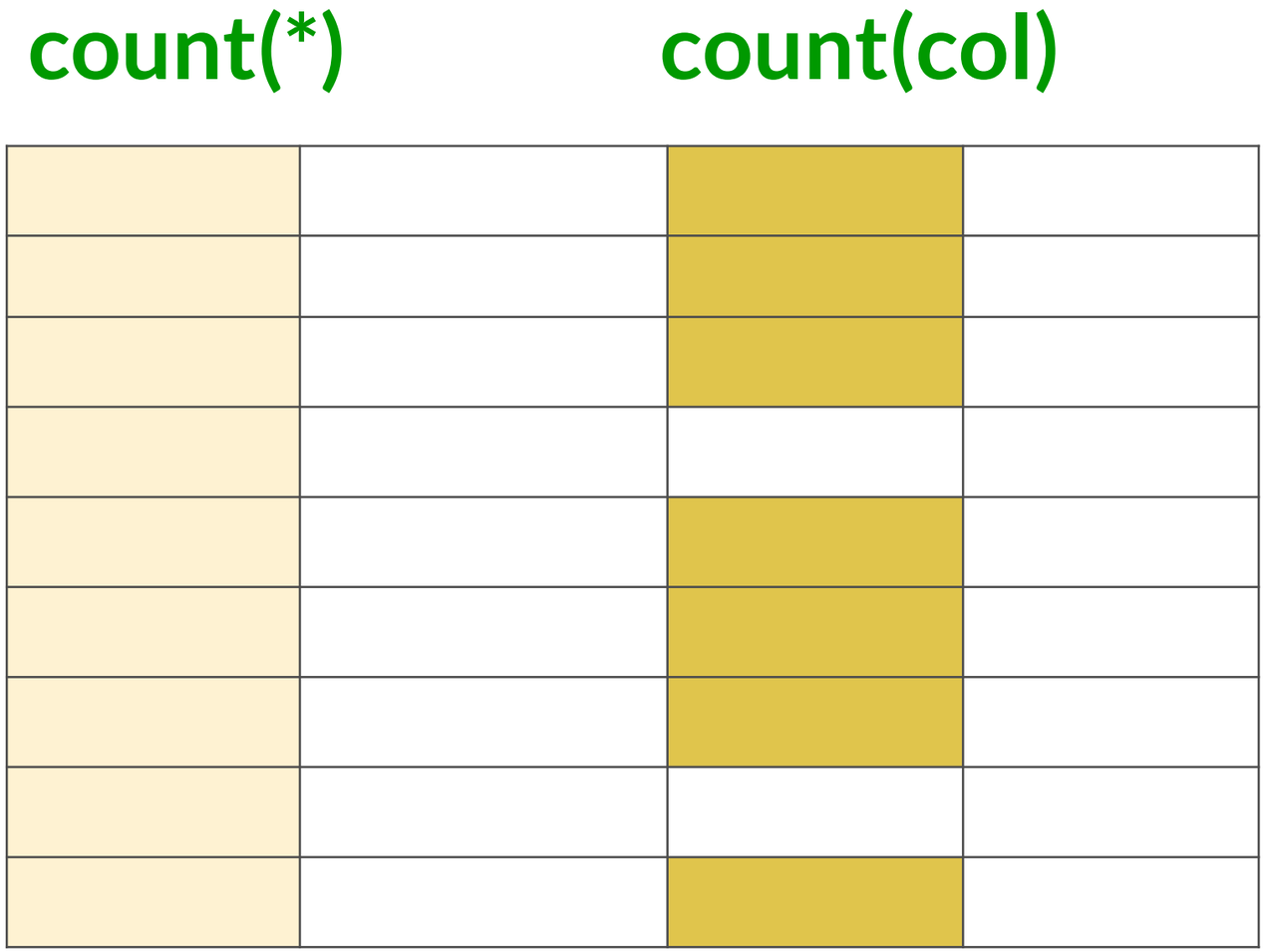
- Aggregate functions ***ignore*** Nulls.

Aggregate Functions

- 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.

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

Aggregate Functions



- 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)

Aggregate Functions

- 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(*) 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)
```

Aggregate Functions

- `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.

Aggregate Functions

```
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) t
    group by country;
```

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

Aggregate Functions

- How many people are both actors and directors?

credits

Aggregate Functions

movie_id	people_id	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

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

- There is no restriction such as “that have played in a movie that they have directed”, so the `movie_id` is irrelevant.
- But if we remove the `movie_id`, we have tons of duplicates. Not a relation!

Aggregate Functions

- People who appear twice are the ones we want.

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

people_id	credited_as
11	D
11	A
12	A
15	A
16	A
17	A
37	D
38	A
39	A

- `distinct` will remove duplicates and provide a true relation.
- We specify the values for `credited_as`
    - There are no other values now
    - but you can't predict the future. Someday there may be producers or directors of photography (cinematographer).

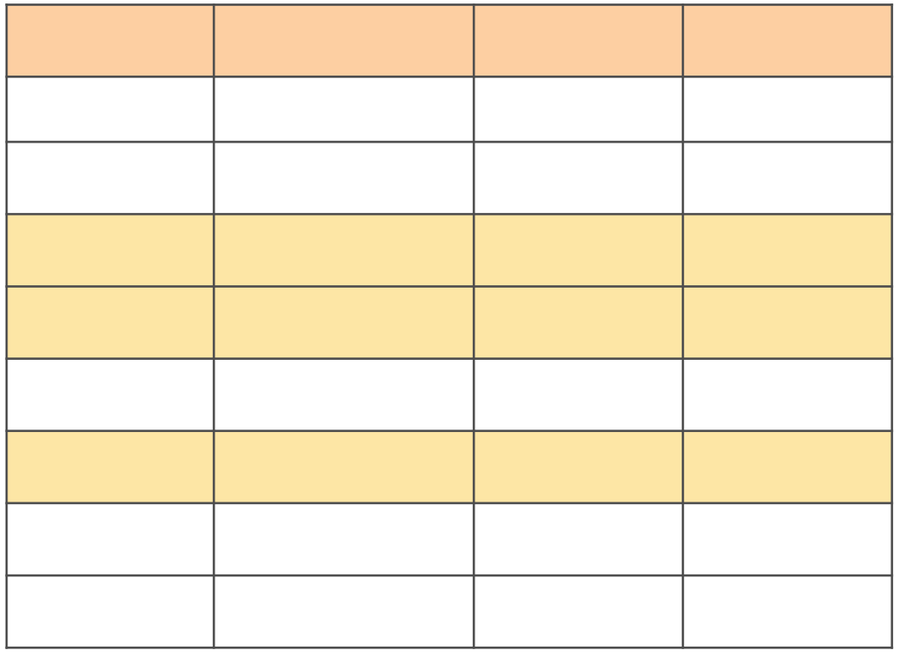
Aggregate Functions

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

```
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;
```

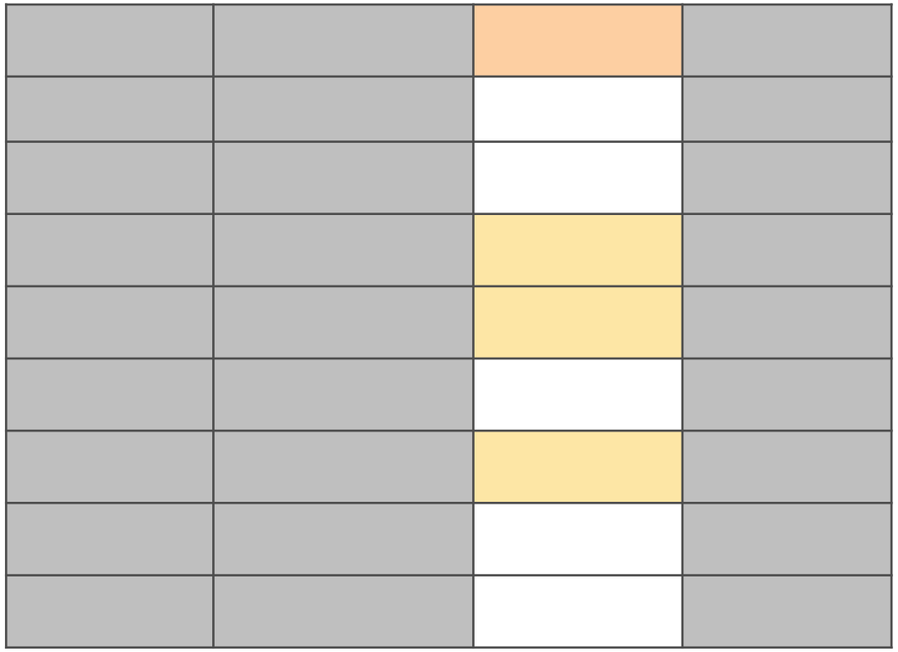
Join

Retrieving Data from Multiple Tables



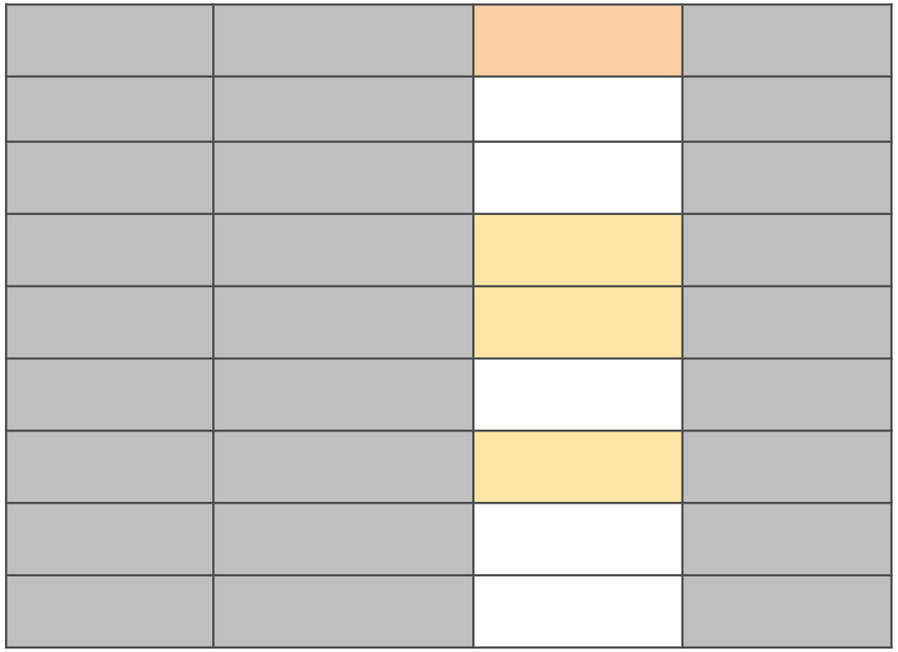
- We have seen the basic operation consisting in filtering rows (an operator called SELECT by Codd)

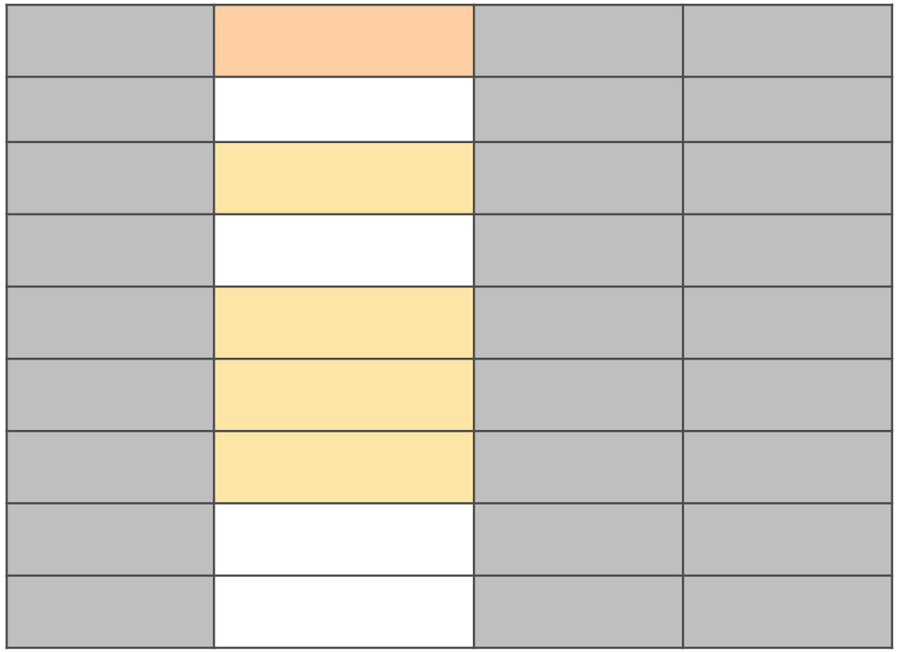
Retrieving Data from Multiple Tables



- 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.

Retrieving Data from Multiple Tables

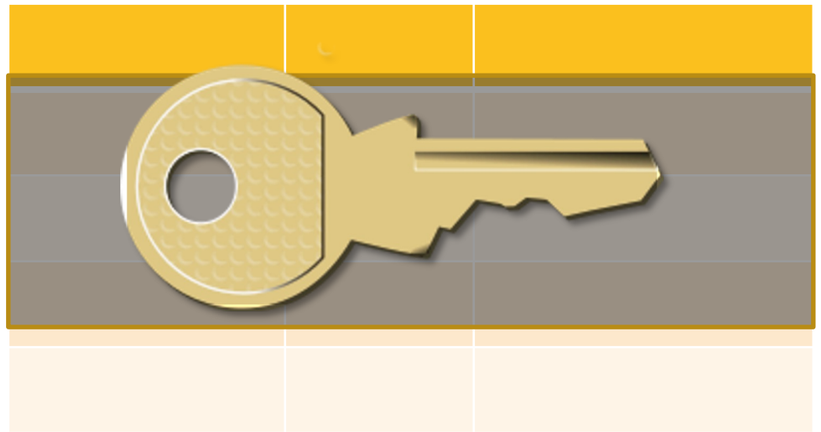




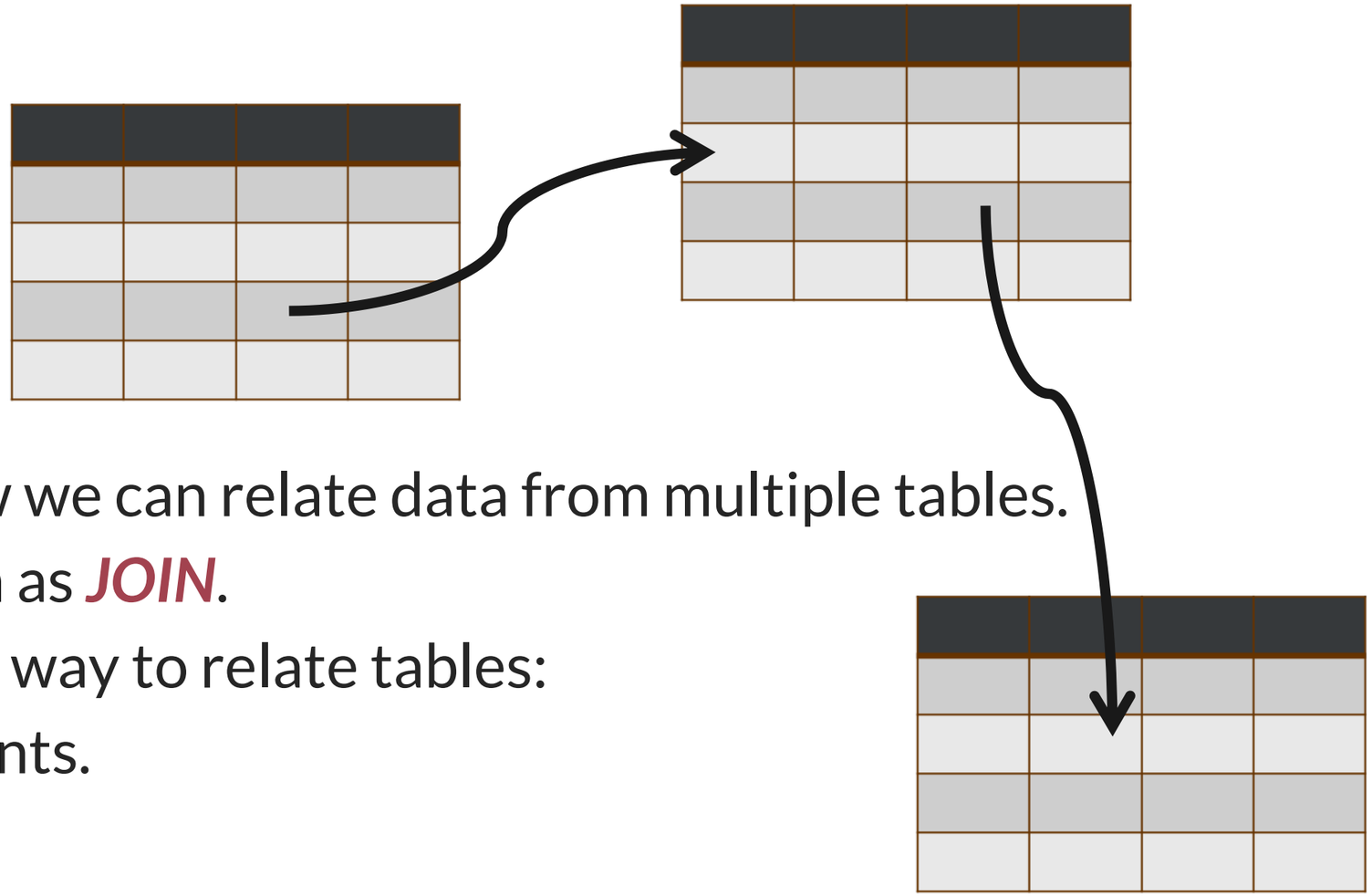
- We have also seen how we can return data that doesn't exist as such in tables by applying functions to columns.

Retrieving Data from Multiple Tables

- What is 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.



Retrieving Data from Multiple Tables



- It's time now to see how we can relate data from multiple tables.
- This operation is known as ***JOIN***.
- We have already seen a way to relate tables:

  foreign key constraints.

Retrieving Data from Multiple Tables

movieid	title	country	year_released
1	Casab	us	1942
2	Goodfellas	us	1990
3	Bronenosets Potyomkin	ru	1925
4	Blade Runner	us	1982
5	Annie Hall	us	1977

country_code	country_name	continent
ru	Russia	Europe
us	United States	America
in	India	Asia
gb	United Kingdom	Europe

- The “country” column in “movies” can be used to retrieve the country name from “countries”.

Retrieving Data from Multiple Tables

- 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;
```

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

Retrieving Data from Multiple Tables

- 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.

movies join countries

movieid	title	country	year_released	country_code	country_name	continent
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	ru	Russia	Europe

Retrieving Data from Multiple Tables

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

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

Retrieving Data from Multiple Tables

`movies join countries`

movieid	title	country	year_released	country_code	country_name	continent
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	ru	Russia	Europe

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

Retrieving Data from Multiple Tables

- From this virtual table
  - Retrieve some columns and apply filtering conditions to any column

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

movieid	title	country	year_released	country_code	country_name	continent
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

Natural Join

- What if we don’t specify the column?
  - Natural join

```sql
select * from people natural join credits;

-- The same as:
select *
from people join credits
on people.peopleid = credits.peopleid;
```

Natural Join

- What if we don’t specify the column?
  - Natural join
- “*If a column has the same name, then we should join on it*”
  - Bad idea!
  - Same name != Same meaning

```
select * from people natural join credits;

-- The same as:
select *
from people join credits
on people.peopleid = credits.peopleid;
```

Natural Join

- What if we don't specify the column?
  - Natural join
- "*If a column has the same name, then we should join on it*"
  - Bad idea!
  - Same name != Same semantic
- In join (not natural join):
  - Use `using` to specify the column with the same name

```sql
select * from people natural join credits;

-- The same as:
select
from people join credits
on people.peopleid = credits.peopleid;

-- Or use "using"
select *
from people join credits using(peopleid);
```

(Maybe) A Good Practice in Writing Queries

- It is preferred not to depend on how database designers name their columns
  - It can be a good practice to use a single (and sometimes straightforward) syntax that works all the time

Keep it simple stupid

```sql
-- Natural join (can sometimes be dangerous)
select * from people natural join credits;

-- The same as:
select *
from people join credits
on people.peopled = credits.peopleid;

-- Or use "using"
select *
from people join credits using(peopleid);

-- A better practice: just write all of them in a unified way
select
from people join credits
on people. peopled = credits.peopleid;
```

Self Join

- Join the same table together
  - For example: How can we find all the pairs of people with the same first name?

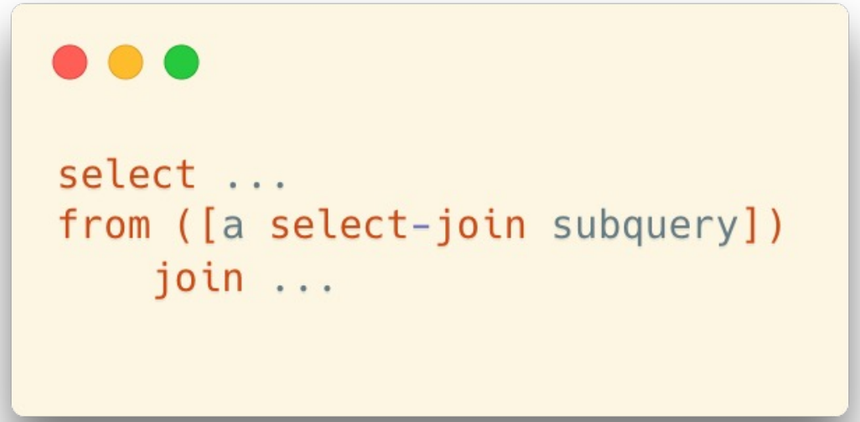
Self Join

- Join the same table together
  - For example: How can we find all the pairs of people with the same first name?

```sql
select *
from people p1 join people p2  -- rename the tables, or you cannot refer to them respectively
on p1.first_name = p2.first_name  -- p1=the first people table; p2=the second people table
where p1.peopleid <> p2.peopleid;  -- remember to filter out the rows with the same person
```

Join in a Subquery

- 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



Chaining Joins Together

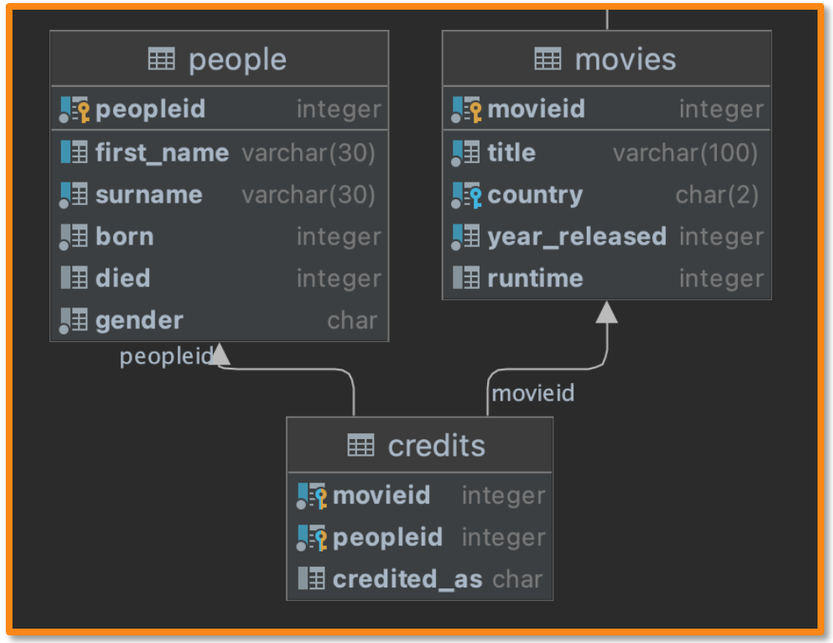
- We can also chain joins the same way we chain filtering conditions with AND.
  - Joins between 10 or 15 tables aren’t uncommon, and queries generated by programs often do much worse.

Chaining Joins Together

- 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.
  - Example: Show names of actors and directors for Chinese movies

Chaining Joins Together

- 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.
  - Example: Show names of actors and directors for Chinese movies



Chaining Joins Together

- 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.
  - Example: Show names of actors and directors for Chinese movies

```
select m.title, c.credited_as, p.first_name, p.surname
from
     movies m join credits c on m.movieid = c.movieid join people p on c.peopleid = p.peopleid
where m.country = 'cn';
```

The Old Way of Writing Joins

- Use commas to separate the tables
  - Example: The solution for the same question in the previous slide
- A little bit history:
  - `join` was introduced in SQL-1999 (later than this original way)
- Relationship to the relational algebra
  - Filtering based on the Cartesian product
    - movies × credits × people

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

The Old Way of Writing Joins

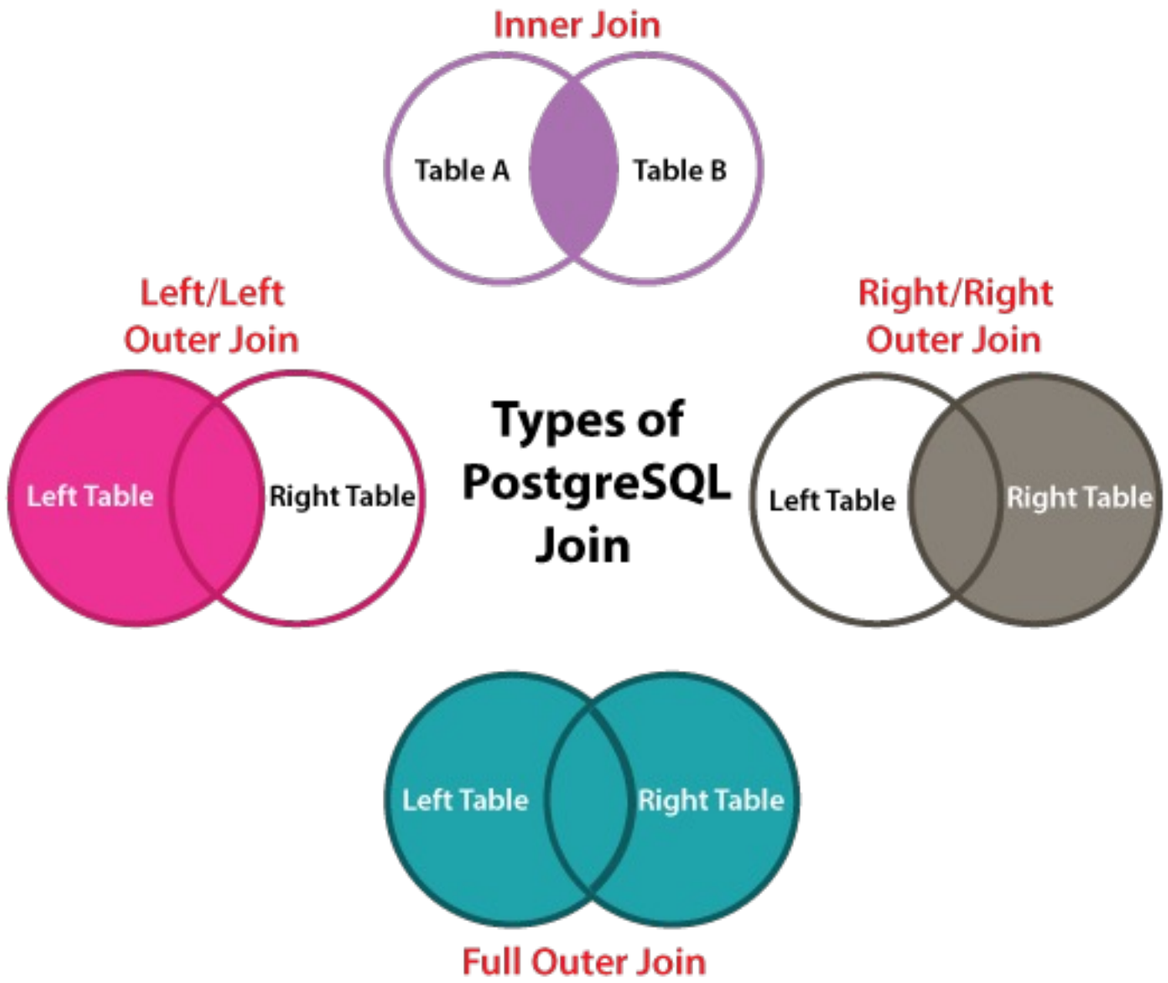
- Problems in the old way:
  - If you forget a comma, it will still work sometimes (interpreted as “renaming”)
  - The semantic meaning of the `where` clause here is a little bit different from the `where` we introduced before
    - (join key vs. filtering condition)
    - If you forget `where`, the query will not return an error but to end up with **HUGE** amount of rows
      - `#movies * #credits * #people`

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

Inner and Outer Joins

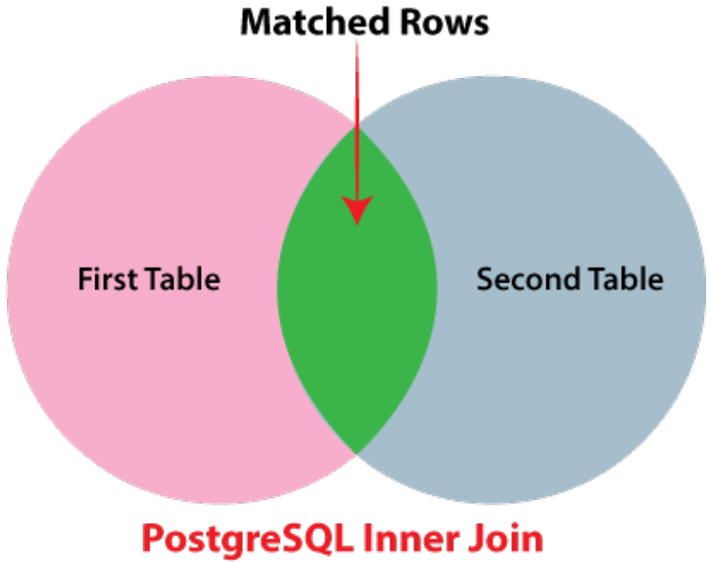
- So far, we only consider the rows with matching values on the corresponding columns
  - However, there are more things you can do with join

Inner and Outer Joins



Inner and Outer Joins

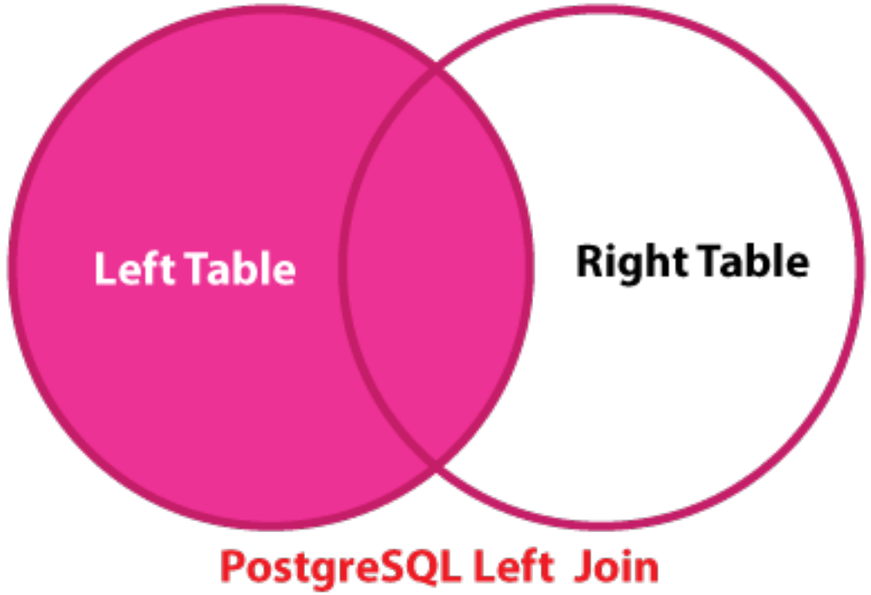
- Inner join
  - The default join type
  - Acturally, all examples before are considered inner joins
  - Only joined rows with matching values are selected




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

Inner and Outer Joins

- Left outer join
  - All the matching rows will be selected
  - … and the rows in the **left table** with **no matches** will be **selected** as well




```
select columns
from table1
LEFT [OUTER] join table2
on table1.column = table2.column;
```

Inner and Outer Joins

- Left outer join
  - Example: there is a movie in 2018 where there is no credit information
    - #9203 (A Wrinkle in Time)

```
select * from movies where movieid = 9203;
```

	movieid	title	country	year_released	runtime
1	9203	A Wrinkle in Time	us	2018	109

Inner and Outer Joins

- Left outer join
  - Example: there is a movie in 2018 where there is no credit information
    - #9203 (A Wrinkle in Time)
    - Inner join of all 2018 movies will not show any matching results for that movie

```
select *
from movies m join credits c
on m.movieid = c.movieid
where m.year_released = 2018;
```

	m.movieid	title	country	year_released	runtime	c.movieid	peopleid	credited_as
1	8987	Red Sparrow	us	2018	145	8987	4062	A
2	8987	Red Sparrow	us	2018	145	8987	6711	A
3	8987	Red Sparrow	us	2018	145	8987	8308	D
4	8987	Red Sparrow	us	2018	145	8987	8310	A
5	8987	Red Sparrow	us	2018	145	8987	11247	A
6	8987	Red Sparrow	us	2018	145	8987	12048	A
7	8987	Red Sparrow	us	2018	145	8987	13071	A
8	8988	Ready Player One	us	2018	0	8988	2934	A
9	8988	Ready Player One	us	2018	0	8988	9819	A
10	8988	Ready Player One	us	2018	0	8988	9971	A
11	8988	Ready Player One	us	2018	0	8988	11390	A
12	8988	Ready Player One	us	2018	0	8988	12758	A
13	8988	Ready Player One	us	2018	0	8988	13421	A
14	8988	Ready Player One	us	2018	0	8988	13850	D
15	8989	Guernsey	gb	2018	0	8989	1864	A
16	8989	Guernsey	gb	2018	0	8989	5280	A
17	8989	Guernsey	gb	2018	0	8989	6523	A
18	8989	Guernsey	gb	2018	0	8989	6836	A
19	8989	Guernsey	gb	2018	0	8989	10643	D
20	8989	Guernsey	gb	2018	0	8989	11261	A
21	8989	Guernsey	gb	2018	0	8989	11733	A
22	8989	Guernsey	gb	2018	0	8989	15708	A
23	8990	A Star Is Born	us	2018	0	8990	2431	A
24	8990	A Star Is Born	us	2018	0	8990	2759	A
25	8990	A Star Is Born	us	2018	0	8990	2939	A
26	8990	A Star Is Born	us	2018	0	8990	2939	D
27	8990	A Star Is Born	us	2018	0	8990	4158	A
28	8990	A Star Is Born	us	2018	0	8990	8105	A
29	8992	Mary Queen of Scots	us	2018	0	8992	272	A
30	8992	Mary Queen of Scots	us	2018	0	8992	2879	A
31	8992	Mary Queen of Scots	us	2018	0	8992	3056	A
32	8992	Mary Queen of Scots	us	2018	0	8992	3365	A
33	8992	Mary Queen of Scots	us	2018	0	8992	8892	A
34	8992	Mary Queen of Scots	us	2018	0	8992	11371	A
35	8992	Mary Queen of Scots	us	2018	0	8992	12435	A
36	8992	Mary Queen of Scots	us	2018	0	8992	12563	A
37	8992	Mary Queen of Scots	us	2018	0	8992	12636	D
38	8993	The Girl in the Spider's Web	se	2018	0	8993	4696	A
39	8993	The Girl in the Spider's Web	se	2018	0	8993	5543	A
40	8993	The Girl in the Spider's Web	se	2018	0	8993	16462	D
41	9202	Black Panther	us	2018	134	9202	3933	A
42	9202	Black Panther	us	2018	134	9202	5588	A
43	9202	Black Panther	us	2018	134	9202	15870	A

Inner and Outer Joins

- Left outer join
  - Example: there is a movie in 2018 where there is no credit information
    - #9203 (A Wrinkle in Time)
    - Inner join of all 2018 movies will not show any matching results for that movie
    - But, left (outer) join can give you a record for the movie (in the left table) where all right-table columns are null

Pay attention to the syntax:

- `left join` or `left outer join`
- But some databases recognize the `outer` keyword, some do not. Refer to the database manual if you meet any error.

```
select * from movies m left join credits c on m.movieid = c.movieid
where m.year_released = 2018;
```

	m.movieid	title	country	year_released	runtime	c.movieid	peopleid	credited_as
41	9202	Black Panther	us	2018	134	9202	3933	A
42	9202	Black Panther	us	2018	134	9202	5588	A
43	9202	Black Panther	us	2018	134	9202	15870	A
44	9203	A Wrinkle in Time	us	2018	109	<null>	<null>	<null>

Inner and Outer Joins

- Left outer join
  - Why? Why should we show the records in the left table with no matches?

  - Scenario: Movie Website (Douban, for example)
    - We cannot just ignore the movies with no credit information
    - Instead, we should list them and also show that credit information is missing
    - All things can be done in a single query
      - And we can distinguish between them by checking the values in the right-table columns

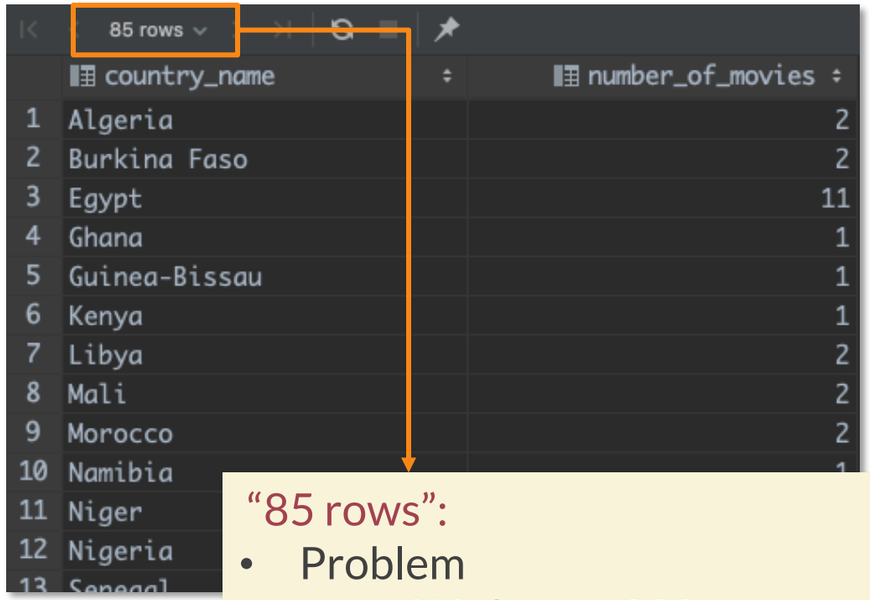
Inner and Outer Joins

- Left outer join
  - Another example: let's count how many movies we have per country (again)

Inner and Outer Joins

- Left outer join
  - Another example: let's count how many movies we have per country (again)

```
select c.country_name, number_of_movies
from countries c join (
    select country as stat_country_code,
        count(*) as number_of_movies
    from movies
    group by country
) stat
on c.country_code = stat_country_code;
```



"85 rows":

- Problem
  - We have ~200 countries in total
  - How can we show the other countries?

Inner and Outer Joins

- Left outer join
  - All countries are here now
  - In addition, how can we replace nulls?

```
select c.country_name, number_of_movies
from countries c left join (
    select country as stat_country_code,
        count(*) as number_of_movies
    from movies
    group by country
) stat
on c.country_code = stat_country_code;
```

185 rows

	country_name	number_of_movies
1	Algeria	2
2	Angola	<null>
3	Benin	<null>
4	Botswana	<null>
5	Burkina Faso	2
6	Burundi	<null>
7	Cameroon	<null>
8	Central African Republic	<null>
9	Chad	<null>
10	Comoros	<null>
11	Congo Brazzaville	<null>
12	Congo Kinshasa	<null>
13	Cote d'Ivoire	<null>
14	Djibouti	<null>
15	Egypt	11
16	Equatorial Guinea	<null>
17	Eritrea	<null>

Inner and Outer Joins

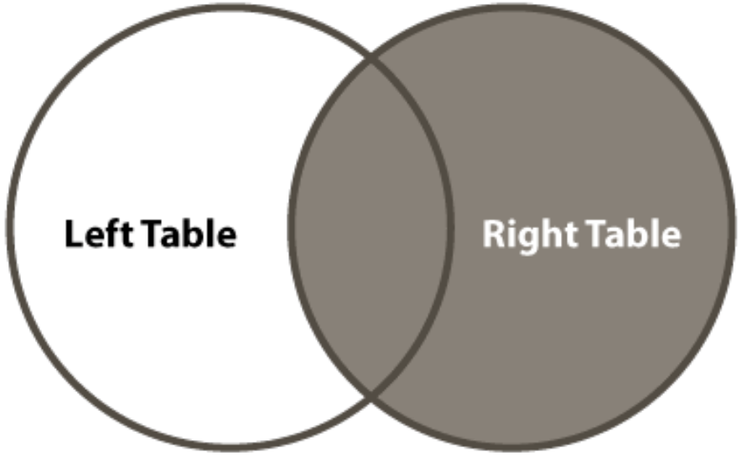
- Left outer join
  - All countries are here now
  - In addition, how can we replace nulls?
    - Add another CASE condition

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

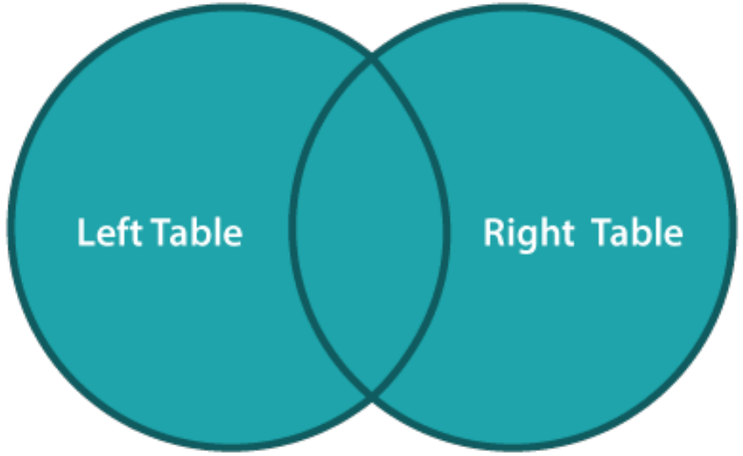
	country_name	number_of_movies
1	Algeria	2
2	Angola	0
3	Benin	0
4	Botswana	0
5	Burkina Faso	2
6	Burundi	0
7	Cameroon	0
8	Central African Republic	0
9	Chad	0
10	Comoros	0
11	Congo Brazzaville	0
12	Congo Kinshasa	0
13	Cote d'Ivoire	0
14	Djibouti	0
15	Egypt	11
16	Equatorial Guinea	0
17	Eritrea	0
18	Ethiopia	0

Inner and Outer Joins

- 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 (by swapping the order of tables in the join list)
    - A full outer join is seldom used



PostgreSQL Right Join



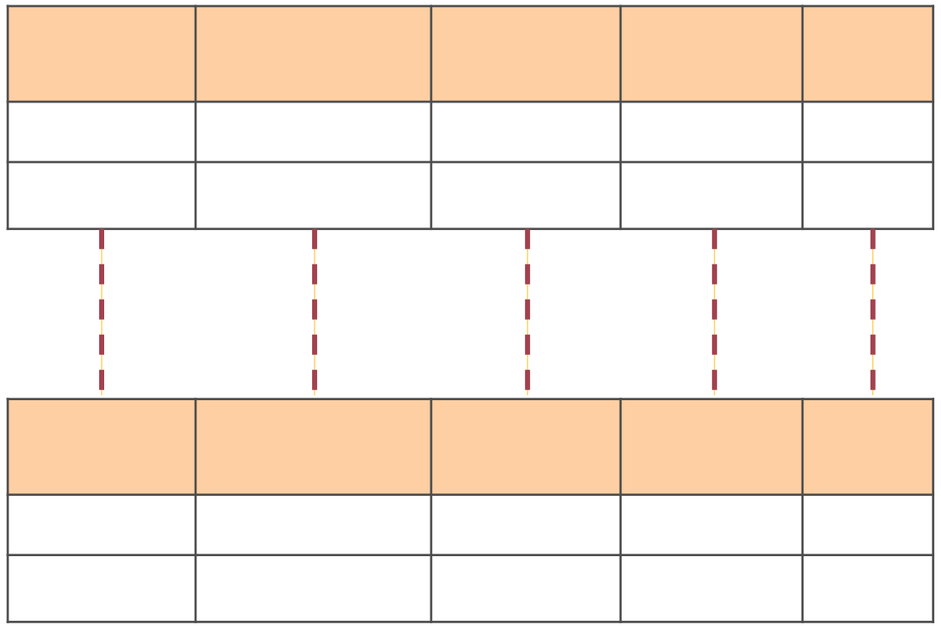
PostgreSQL Full Outer Join

Set Operators

Set Operators

- Union
  - Takes two result sets and combines them into a single result set

- Union requires two (commonsensical) conditions:
  - They must return the same number of columns
  - The data types of corresponding columns must match.



Set Operators

- Union
  - Example: Stack US and GB movies together

```sql
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;
```

	movieid	title	year_released	country
1	3840	The Secret Life of Walter Mitty	1947	us
2	678	The Ox-Bow Incident	1943	us
3	3174	The Red House	1947	us
4	5152	Minesweeper	1943	us
5	1487	Kiss of Death	1947	us
6	3408	Ministry of Fear	1944	us
7	2543	The Way to the Stars	1945	gb
8	5341	All Through the Night	1942	us
9	1435	They Live by Night	1948	us
10	2644	Criminal Court	1946	us
11	7250	The Seventh Veil	1945	gb
12	7341	Mr. Lucky	1943	us

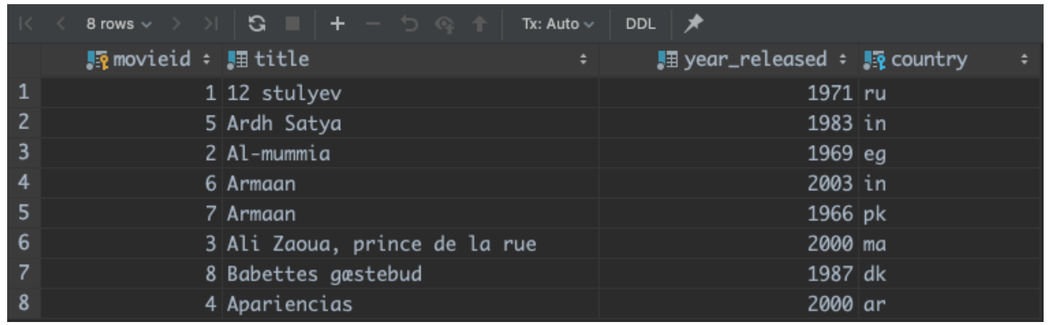
Set Operators

- Union
  - Usage scenario: combine movies from two tables, one for standard accounts and one for VIP accounts
    - We don’t want to miss the “standard movies” for the VIP accounts

Set Operators

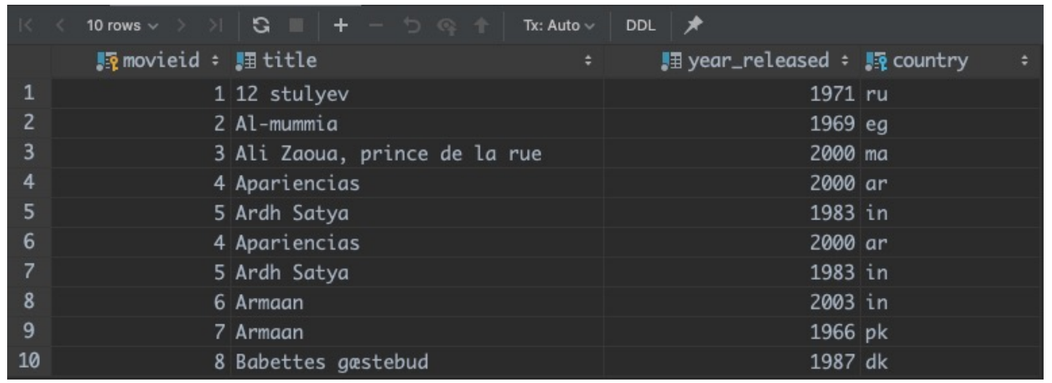
- Union
  - Warning: `union` will remove duplicated rows
    - Instead, you can use `union all`

```
(select movieid, title, year_released, country
from movies limit 5 offset 0)
union
(select movieid, title, year_released, country
from movies limit 5 offset 3);
```



	movieid	title	year_released	country
1	1	12 stulyev	1971	ru
2	5	Ardh Satya	1983	in
3	2	Al-mummia	1969	eg
4	6	Armaan	2003	in
5	7	Armaan	1966	pk
6	3	Ali Zaoua, prince de la rue	2000	ma
7	8	Babettes gæstebud	1987	dk
8	4	Apariencias	2000	ar

```
(select movieid, title, year_released, country
from movies limit 5 offset 0)
union all
(select movieid, title, year_released, country
from movies limit 5 offset 3);
```



	movieid	title	year_released	country
1	1	12 stulyev	1971	ru
2	2	Al-mummia	1969	eg
3	3	Ali Zaoua, prince de la rue	2000	ma
4	4	Apariencias	2000	ar
5	5	Ardh Satya	1983	in
6	4	Apariencias	2000	ar
7	5	Ardh Satya	1983	in
8	6	Armaan	2003	in
9	7	Armaan	1966	pk
10	8	Babettes gæstebud	1987	dk

Set Operators

- Intersect (`intersect`)
  - Return the rows that appears in both tables
- Except (`except`)
  - Return the rows that appear in the first table but not the second one
  - Sometimes written as `minus` in some database products

- However, they are not used as much as union
  - intersect -> inner join
  - except -> left outer join with an “is null” condition

Subquery

Subquery

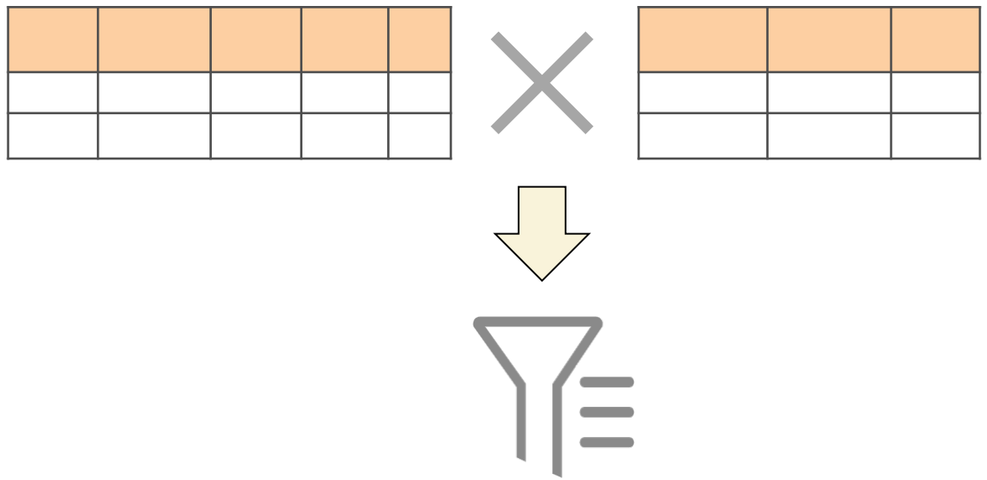
- We have used subqueries after `from` before
  - … in order to build queries upon a query result

- And, we can add subqueries after `select` and `where` as well

Subquery after Select

- Example: show titles, released years, and country names for non-US movies
  - Solution 1: Join

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



Subquery after Select

- Example: show titles, released years, and country names for non-US movies
  - Solution 2: Nested selection

```
select m.title,
       m.year_released,
       m.country
from movies m
where m.country <> 'us';
```

… still a country code though

- How can we replace it with the country name?

Subquery after Select

- Example: show titles, released years, and country names for non-US movies
  - Solution 2: Nested selection

```sql
select m.title,
       m.year_released,
       m.country
from movies m
where m.country <> 'us';
```

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

A subquery after select:

- For each selected row in the outer query, find the corresponding country name in the `countries` table

Subquery after Where

- Recall: the `in()` operator
  - It can be used as the equivalent for a series of equalities with OR (it has also other interesting uses)
  - It may make a comparison clearer than a parenthesized expression

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

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

Subquery after Where

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

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

Subquery after Where

- Example: Select all European movies
  - How can we specify the filtering condition?

```
select country,
       year_released,
       title
from movies
where [?]
```

Subquery after Where

- Example: Select all European movies
  - A horrible solution: list all European countries with `or`

```
select country,
       year_released,
       title
from movies
where country = 'fr' or country = 'de' or ...
```

Subquery after Where

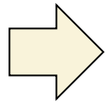
- Example: Select all European movies
  - A (slightly better) solution: list all European countries in an `in` operator

```
select country,
       year_released,
       title
from movies
where country in('fr', 'de', ...)
```

Subquery after Where

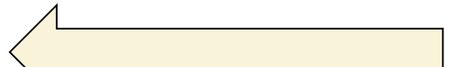
- Example: Select all European movies
  - A (slightly better) solution: list all European countries in an `in` operator

```
select country,
       year_released,
       title
from movies
where country in('fr', 'de', ...)
```



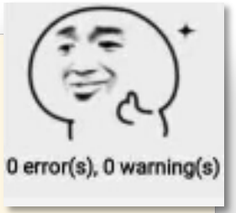
```
select * from countries where continent = 'EUROPE';
```

40 rows



Subquery after Where

- Example: Select all European movies
  - A proper solution: (dynamically) fill in the list of country codes in an `in` operator




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

```sql
select country,
       year_released,
       title
from movies
where country in('fr', 'de', ...)
```

The same results (if you fill in all European country codes on the right side)

- But you can automatically generate this list
- Especially useful when the table in the subquery changes often

Subquery after 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.

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

Subquery after Where

- Some important points for `in()`
  - `in()` means an implicit distinct in the subquery
    - `in('cn', 'us', 'cn', 'us', 'us')` is equal to `in('cn', 'us')`

Subquery after Where

- Some important points for `in()`
  - `in()` means an implicit distinct in the subquery
    - `in('cn', 'us', 'cn', 'us', 'us')` is equal to `in('cn', 'us')`
  - null values in `in()`
    - Be extremely cautious if you are using `not in(...)` with a null value in it

Subquery after Where

- Some important points for `in()`
  - `in()` means an implicit distinct in the subquery
    - `in('cn', 'us', 'cn', 'us', 'us')` is equal to `in('cn', 'us')`
  - null values in `in()`
    - Be extremely cautious if you are using `not in(...)` with a null value in it

```
value not in(2, 3, null)
⇒not (value=2 or value=3 or value=null)
⇒value<>2 and value<>3 and value<>null
⇒false  -- always false or null, never true
```

… however, `value=null` and `value<>null` are always not true:

- We should use `is [not] null` instead

Thus, the `not in()` expression always returns false, and hence no row will be selected and returned.

More about NULL

Expressions with NULL Values

- Most expressions with `NULL` will be evaluated into `NULL`
  - Arithmetic operations:

```
col+NULL -> NULL
col-NULL -> NULL
col*NULL -> NULL
col/NULL -> NULL

...
```

  - Comparison operations:

```
(col > NULL) -> NULL
(col = NULL) -> NULL

...
```

Expressions with NULL Values

- Most expressions with `NULL` will be evaluated into `NULL`
  - But, there are some conditions where the values are not `NULL`

```
TRUE and NULL -> NULL
FALSE and NULL -> FALSE

TRUE or NULL -> TRUE
FALSE or NULL -> NULL
```

Logical operators (or, and):

- Three-valued logic (true, false, and unknown)

More on this: Three-valued logic and its application in SQL
`https://en.wikipedia.org/wiki/Three-valued_logic#SQL`

```
col is NULL -> True or False
```

The way we use to check a `NULL` value: use `is`, not `=`

Recall: Subquery after Where

- Some important points for `in()`
  - `in()` means an implicit distinct in the subquery
    - `in('cn', 'us', 'cn', 'us', 'us')` is equal to `in('cn', 'us')`
  - null values in `in()`
    - Be extremely cautious if you are using `not in(...)` with a null value in it

```
value not in(2, 3, null)
⇒not (value=2 or value=3 or value=null)
⇒value<>2 and value<>3 and value<>null
⇒false  -- always false or null, never true
```

- If value is 2, the result is:
  `TRUE and FALSE and NULL -> FALSE`
- if value is 5, the result is:
  `TRUE and TRUE and NULL -> NULL`
- if value is NULL, the result is :
  `NULL and NULL and NULL -> NULL`

Ordering

Ordering in SQL

- `order by`
  - A simple expression in SQL to order a result set
  - It comes at the end of a query
    - ... and, you can have it in subqueries, definitely
  - Followed by the list of columns used as sort columns

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

	title	year_released
1	Ben Hur	1907
2	The Lonely Villa	1909
3	From the Manger to the Cross	1912
4	Falling Leaves	1912
5	Traffic in Souls	1913
6	At Midnight	1913
7	Lime Kiln Field Day	1913
8	The Sisters	1914
9	The Only Son	1914
10	Tess of the Storm Country	1914
11	Under the Gaslight	1914
12	Brute Force	1914
13	The Wishing Ring: An Idyll of Old England	1914

Ordering in SQL

- No matter how difficult the query is, you can apply order by to any result set

```
select m.title,
       m.year_released
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)
order by m.year_released
```

	title	year_released
1	Snehaseema	1954
2	Nairu Pidicha Pulivalu	1958
3	Mudiyanaya Puthran	1961
4	Puthiya Akasam Puthiya Bhoomi	1962
5	Doctor	1963
6	Aadyakiranangal	1964
7	Odayil Ninnu	1965
8	Adimakal	1969
9	Karakanakadal	1971
10	Ghatashraddha	1977
11	Kramer vs. Kramer	1979
12	The Champ	1979
13	The Shining	1980

Ordering in SQL

- Ordering with joins
  - We can sort by any column of any table in the join (remember the super wide table with all the columns from all tables involved)

```sql
select c.country_name,
       m.title,
       m.year_released
from movies m
     inner join countries c
     on c.country_code = m.country
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)
order by m.year_released
```

	country_name	title	year_released
1	India	Snehaseema	1954
2	India	Nairu Pidicha Pulivalu	1958
3	India	Mudiyanaya Puthran	1961
4	India	Puthiya Akasam Puthiya Bhoomi	1962
5	India	Doctor	1963
6	India	Aadyakiranangal	1964
7	India	Odayil Ninnu	1965
8	India	Adimakal	1969
9	India	Karakanakadal	1971
10	India	Ghatashraddha	1977
11	United States	Kramer vs. Kramer	1979
12	United States	The Champ	1979
13	United States	The Shining	1980

Ordering in SQL

- Ordering with joins
  - We can sort by any column of any table in the join (remember the super wide table with all the columns from all tables involved)

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

	country_name	title	year_released
1	India	Snehaseema	1954
2	India	Nairu Pidicha Pulivalu	1958
3	India	Mudiyanaya Puthran	1961
4	India	Puthiya Akasam Puthiya Bhoomi	1962
5	India	Doctor	1963
6	India	Aadyakiranangal	1964
7	India	Odayil Ninnu	1965
8	India	Adimakal	1969
9	India	Karakanakadal	1971
10	India	Ghatashraddha	1977
11	United States	Kramer vs. Kramer	1979
12	United States	The Champ	1979
13	United States	The Shining	1980

Advanced Ordering

- Multiple columns
  - For example:
    - The result set will be order by `col1` first
    - If there are rows with the same value on `col1`, these rows will be ordered by `col2`.

```
order by col1, col2, ...
```

- Ascending or descending order
  - Add `desc` or `asc` after the column
    - However, `asc` is the default option and thus always omitted

```
-- Order col1 descendingly
order by col1 desc

-- Order based on col1 first, then col2.
-- col1 will be in the descending order, col2 ascending.
order by col1 desc, col2 asc, ...
```

Advanced Ordering

- Self-defined ordering
  - Use "`case` … `when`" in `order by` to define criteria on how to order the rows

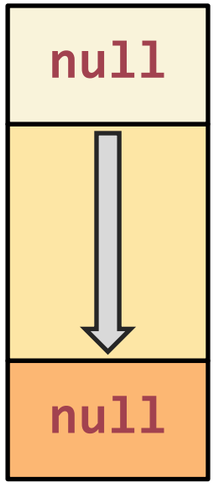
```
select * from credits
order by
    case credited_as
        when 'D' then 1
        when 'A' then 2
    end desc;
```

Data Types in Ordering

- Ordering depends on the data type
  - Strings: alphabetically,
  - Numbers: numerically
  - Dates and times: chronologically

Data Types in Ordering

- What about `NULL`?
  - It is implementation-dependent
  - SQL Server, MySQL and SQLite:
    - “nothing” is smaller than everything
  - Oracle and PostgreSQL:
    - “nothing” is greater than anything



Ordering in Text Data

- Remember, we have many different languages other than English
  - “Alphabetical order” in different languages means different things
    - Mandarin: Pinyin? Number of strokes?
    - Swedish and German
      - “ö” is considered the last letter in Swedish, while in German it is ordered after “o”.
  - Collation

Self Study: Text Encoding

- Key Question: How does characters represented in a computer?
  - Wikipedia – Character encoding: `https://en.wikipedia.org/wiki/Character_encoding`
  - A video on Bilibili: `https://www.bilibili.com/video/BV1xP4y1J7CS`

Self Study: Text Encoding

手持两把锟斤拷，
口中疾呼烫烫烫。
脚踏千朵屯屯屯，
笑看万物锘锘锘。

- Try to answer the following questions:
  - What are ASCII, Unicode, UTF-8, and UTF-16? What are the relationships between them?
  - What are GB2312, GB18030, and GBK? What are “锟斤拷” and “烫烫烫”? How can you make it (not) happen?
  - Given a string with several characters, can you print the bitmap of this string?
  - Are emojis characters? How can you insert an emoji in a text editor?
  - What are the default character encodings in different platforms?
    - OS: Windows, MacOS, Linux
    - DBMS: PostgreSQL, etc.
    - Programming Languages: Java, C, C++, Python, etc.
  - How can we translate strings from one encoding to another?
    - E.g., with text editors (Windows Notepad, VSCode, Sublime Text, etc.); in programming languages; in DBMS

Limit and Offset

- Get a slice of the long query result
  - `limit k offset p`
    - Return the top-k rows in the result set and skip the first p rows
    - `offset` is optional (which means "`offset 0`")
  - Always used together with order by
    - E.g., get the top-k query results under a certain ordering criteria
  - * In some DBMS, the syntax can be different
    - Always refer to the software manual for specific features

```
select * from movies
where country = 'us'
order by year_released
limit 10 offset 5
```

```
select * from movies
where country = 'us'
order by year_released
limit 10
```

Window Function

Scalar Functions and Aggregation Functions

- Scalar function
  - Functions that operate on values in the current row
    - Recall: “Some Functions”, Lecture 3

```
round(3.141592, 3)  -- 3.142
trunc(3.141592, 3)  -- 3.141
```

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

- Aggregation function
  - Functions that operate on sets of rows and return an aggregated value
    - Recall: “Aggregate Functions”, Lecture 4

```
count(*)/count(col), min(col), max(col), stddev(col), avg(col)
```

Issues with Aggregate Functions

- A Problem: In aggregated functions, the details of the rows are vanished
  - For example: If we ask for the year of the oldest movie per country,
    - … we get a country, a year, and nothing else.

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

Issues with Aggregate Functions

- A Problem: In aggregated functions, the details of the rows are vanished
  - For example: If we ask for the year of the oldest movie per country,
    - … we get a country, a year, and nothing else.

If we want some more details, like the title of the oldest movies for each country, we can only use self-join to keep the columns

- And there is also one more problem in the query on the right side. Can you find it?

```
select m1.country,
       m1.title,
       m1.year_released
from movies m1
     inner join
     (select country,
        min(year_released) minyear
        from movies
        group by country) m2
on m2.country = m1.country and m2.minyear = m1.year_released
order by m1.country
```

Issues with Aggregate Functions

- A Problem: In aggregated functions, the details of the rows are vanished
  - Another example: How can we rank the movies in each country separately based on the released year?
    - “order by” for subgroups
  - One more example: Get the top-3 oldest movies for each country.
    - How can we implement it?

Window Function

- Syntax:

```
<function> over (partition by <col_p> order by <col_o1, col_o2, …>)
```

  - `<function>`: we can apply (1) ranking window functions, or (2) aggregation functions
  - `partition by`: specify the column for grouping
  - `order by`: specify the column(s) for ordering in each group

Ranking Window Function

- Example
  - How can we rank the movies in each country separately based on the released year?
    - “order by” for subgroups

```
select country,
       title,
       year_released,
       rank() over (
           partition by country order by year_released
       ) oldest_movie_per_country
from movies;
```

	country	title	year_released	oldest_movie_per_country
1	am	Sayat Nova	1969	1
2	ar	Pampa bárbara	1945	1
3	ar	Albéniz	1947	2
4	ar	Madame Bovary	1947	2
5	ar	La bestia debe morir	1952	4
6	ar	Las aguas bajan turbias	1952	4
7	ar	Intermezzo criminal	1953	6
8	ar	La casa del ángel	1957	7
9	ar	Bajo un mismo rostro	1962	8
10	ar	Las aventuras del Capitán Piluso	1963	9
11	ar	Savage Pampas	1966	10
12	ar	La hora de los hornos	1968	11
13	ar	Waiting for the Hearse	1985	12
14	ar	La historia oficial	1985	12
15	ar	Hombre mirando al sudeste	1986	14

Ranking Window Function

- Example
  - How can we rank the movies in each country separately based on the released year?
    - "order by" for subgroups

```
select country,
       title,
       year_released,
       rank() over (
           partition by country order by year_released
       ) oldest_movie_per_country
from movies;
```

You can also add "desc" here, similar to the "order by" we introduced before

country	title	year_released	oldest_movie_per_country
ar	some title	1948	1
ar	some title	1959	2
ar	some title	1980	3
cn	some title	1987	1
cn	some title	2002	2
uk	some title	1985	1
uk	some title	1992	2
uk	some title	2010	3

`partition by country`

- the selected rows will be grouped (partitioned) according to the values in the column `country`

`rank()`

- A function to say that "I want to order the rows in each partition"
- No parameters in the parentheses

`order by year_released`

- In each group (partition), the rows will be ordered by the column "`year_released`"

Ranking Window Function

- Example
  - How can we rank the movies in each country separately based on the released year?
    - “order by” for subgroups

```
select country,
       title,
       year_released,
       rank() over (
           partition by country order by year_released
       ) oldest_movie_per_country
from movies;
```

country	title	year_released	oldest_movie_per_country
ar	some title	1948	1
ar	some title	1959	2
ar	some title	1980	3
cn	some title	1987	1
cn	some title	2002	2
uk	some title	1985	1
uk	some title	1992	2
uk	some title	2010	3

Note: partition functions can only be used in the `select` clause

- … since it is designed to work on the query result

Ranking Window Function

- Example
  - How can we rank the movies in each country separately based on the released year?
    - “order by” for subgroups

```
select country,
       title,
       year_released,
       rank() over (
           partition by country order by year_released
       ) oldest_movie_per_country
from movies;
```

country	title	year_released	oldest_movie_per_country
ar	some title	1948	1
ar	some title	1959	2
ar	some title	1980	3
cn	some title	1987	1
cn	some title	2002	2
uk	some title	1985	1
uk	some title	1992	2
uk	some title	2010	3

Partitioned by country
- i.e., a country in a group

An order value is computed for each row in a partition.
- Only inside the partition, not across the entire result set

Ranking Window Function

- Why window function, not group by?
  - “Group by” reduces the rows in a group (partition) into one result, which is the meaning of “aggregation”
    - Then, the values in non-aggregating columns are vanished
  - Window functions do not reduce the rows
    - Instead, they attach computed values next to the rows in a group (partition) and keep the details
    - Actually, the partition here means “window”: an affective range for statistics

Ranking Window Function

- Some more ranking window functions
  - Besides `rank()`, we also have `dense_rank()` and `row_number()`
  - The difference is about how they treat rows with the same rank

```
select country,
       title,
       year_released,

       rank() over (
           partition by country order by year_released
       ) rank_result,

       dense_rank() over (
           partition by country order by year_released
       ) dense_rank_result,

       row_number() over (
           partition by country order by year_released
       ) row_number_result
from movies;
```

country	title	year_released	rank_result	dense_rank_result	row_number_result
cn	some title	1948	1	1	1
cn	some title	1959	2	2	2
cn	some title	1959	2	2	3
cn	some title	1987	4	3	4
cn	some title	2002	5	4	5
uk	some title	1985	1	1	1
uk	some title	1992	2	2	2
uk	some title	2010	3	3	3

Aggregation Functions as Window Functions

- `max(col)` and `min(col)`

```
select country,
       title,
       year_released,
       min(year_released) over (
           partition by country order by year_released
       ) oldest_movie_per_country
from movies;
```

Need to specify a column in the parameter list

	cou...	title	year_released	oldest_movie_per_country
1	am	Sayat Nova	1969	1969
2	ar	Pampa bárbara	1945	1945
3	ar	Albéniz	1947	1945
4	ar	Madame Bovary	1947	1945
5	ar	La bestia debe morir	1952	1945
6	ar	Las aguas bajan turbias	1952	1945
7	ar	Intermezzo criminal	1953	1945
8	ar	La casa del ángel	1957	1945
9	ar	Bajo un mismo rostro	1962	1945
10	ar	Las aventuras del Capitán Piluso	1963	1945
11	ar	Savage Pampas	1966	1945
12	ar	La hora de los hornos	1968	1945
13	ar	Waiting for the Hearse	1985	1945
14	ar	La historia oficial	1985	1945
15	ar	Hombre mirando al sudeste	1986	1945

The min/max value for each partition is assigned for all the rows inside this partition

Aggregation Functions as Window Functions

- `sum(col)`, `count(col)`, `avg(col)`, `stddev(col)`, etc.
  - Different from min/max: for these aggregation functions, it means the aggregation value from the first row to the current row in its partition when `order by` is specified

```
select country,
       title,
       year_released,
       sum(runtime) over (
           partition by country order by year_released
       ) total_runtime_till_this_row
from movies;
```

		title	year_released	total_runtime_till_this_row
1	am	Sayat Nova	1969	78
2	ar	Pampa bárbara	1945	98
3	ar	Albéniz	1947	308
4	ar	Madame Bovary	1947	308
5	ar	La bestia debe morir	1952	494
6	ar	Las aguas bajan turbias	1952	494
7	ar	Intermezzo criminal	1953	494
8	ar	La casa del ángel	1957	570
9	ar	Bajo un mismo rostro	1962	695
10	ar	Las aventuras del Capitán Piluso	1963	785
11	ar	Savage Pampas	1966	897
12	ar	La hora de los hornos	1968	1157
13	ar	Waiting for the Hearse	1985	1354
14	ar	La historia oficial	1985	1354

However, if there is no `order by`, the behavior will be similar to `min()` and `max()`

- One result for all rows

Pay attention to the behavior on rows with the same rank:

- They are “treated like the same row” here

Exercise

- Question: How can we get the top-5 most recent movies for each country?
  - Hint: Use a subquery in the "`from`" clause

Exercise

- Question: How can we get the top-5 most recent movies for each country?
  - Hint: Use a subquery in the "`from`" clause

```
select x.country,
       x.title,
       x.year_released
from (
 select country,
        title,
        year_released,
        row_number()
        over (partition by country
              order by year_released desc) rn
 from movies) x
where x.rn <= 5
```

Update and Delete

So Far…

- We have learned:
  - How to access existing data in tables (select)
  - How to create new rows (insert)

- CRUD/CURD
  - create, read, update, delete
    - In SQL: insert, select, update, delete
    - In RESTful API: Post, Get, Put, Delete
  - Necessary operations for persistent storage

Update

- Make changes to the existing rows in a table
- `update` is the command that changes column values
  - You can even set a non-mandatory column to NULL
  - The change is applied to all rows selected by the `where`

```
update table_name
set column_name = new_value,
    other_col = other_new_val,
    ...
where ...
```

Update

- Remember
  - When you are doing any experiments with writing operations (update, delete), **backup the data first**
    - E.g., copy the tables

Update

- Example: A nobiliary particle is used in a surname or family name in many Western cultures to signal the nobility of a family.
  - We may want to modify some names in such a way as they sort as they should.

23 rows | Tx: Auto | DDL

	peopleid	first_name	surname	born	died	gender
1	16439	Axel	von Ambesser	1910	1988	M
2	16440	Daniel	von Bargen	1950	2015	M
3	16441	Eduard	von Borsody	1898	1970	M
4	16442	Suzanne	von Borsody	1957	<null>	F
5	16443	Tomas	von Brömssen	1943	<null>	M
6	16444	Erik	von Detten	1982	<null>	M
7	16445	Theodore	von Eltz	1893	1964	M
8	16446	Gunther	von Fritsch	1906	1988	M
9	16447	Katja	von Garnier	1966	<null>	F
10	16448	Harry	von Meter	1871	1956	M
11	16449	Jenna	von Oÿ	1977	<null>	F
12	16450	Alicia	von Rittberg	1993	<null>	F
13	16451	Daisy	von Scherler Mayer	1966	<null>	F
14	16452	Gustav	von Seyffertitz	1862	1943	M
15	16453	Josef	von Sternberg	1894	1969	M

John von Neumann

Update

- Example: A nobiliary particle is used in a surname or family name in many Western cultures to signal the nobility of a family.
  - We may want to modify some names in such a way as they sort as they should.

- First, how can we find these names?

Update

- Example: A nobiliary particle is used in a surname or family name in many Western cultures to signal the nobility of a family.
  - We may want to modify some names in such a way as they sort as they should.

- First, how can we find these names?
  - Wildcards

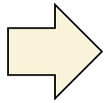
```
select * from people_1 where surname like 'von %';
```

Update

- Example: A nobiliary particle is used in a surname or family name in many Western cultures to signal the nobility of a family.
  - We may want to modify some names in such a way as they sort as they should.

- Then, how should we update the names?

```
(first_name) John
(surname) von Neumann
```



```
(first_name) John
(surname) Neumann (von)
```

  - Try the transformation with select:

```
select replace('von Neumann', 'von ', '') || ' (von)';
```

	?column?
1	Neumann (von)

Update

- Example: A nobiliary particle is used in a surname or family name in many Western cultures to signal the nobility of a family.
  - We may want to modify some names in such a way as they sort as they should.

- Finally, the `update` statement:

```
-- Specify the table
update people

-- Set the update rule
set surname = replace(surname, 'von ', '') || ' (von)'

-- Find the rows that need to be updated
where surname like 'von %';
```

This could be used to postfix all surnames starting by 'von' with '(von)' and turn for instance 'von Stroheim' into 'Stroheim (von)'

Update

- The `where` clause specifies the affected rows
  - However, you can use update without `where`, where the updates will be applied to all rows in the table
    - Use with caution!
    - Sometimes, there will be a warning in IDEs such as DataGrip

Update

- The update operation may not be successful when constraints are violated
  - For example, update the primary key but with duplicated values

```
update people set peopleid = 1 where peopleid < 10;
```

```
[23505] ERROR: duplicate key value violates unique constraint "people_pkey"
Detail: Key (peopleid)=(1) already exists.
```

  - This is why we need constraints when creating tables: avoid unacceptable writing operations that break the integrity of the tables

Update

- Subqueries in update
  - Complex update operations where values are based on a query result

- Example: Add a column in people table to record the number of movies one has joined (either directed or played a role in)

Update

- Example: Add a column in people table to record the number of movies one has joined (either directed or played a role in)
  - First, how do we count the movies for a person?
    - (Used as the subquery part in the update statement)

```
select count(*) from credits c where c.peopleid = [some peopleid];
```

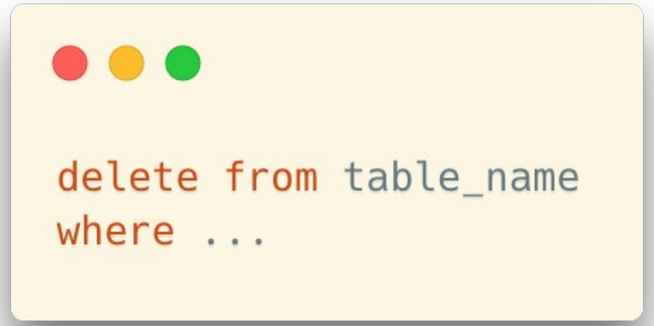
Update

- Example: Add a column in people table to record the number of movies one has joined (either directed or played a role in)
  - First, how do we count the movies for a person?
    - (Used as the subquery part in the update statement)
  - Then, let's update the data

```sql
update people p

set num_movies = (
    select count(*) from credits c where c.peopleid = p.peopleid
)

where peopleid < 500;
-- This where is only for testing purpose;
-- You should change it (or remove it) when in actual use.
```

Delete

- As the name shows, `delete` removes rows from tables



  - If you omit the WHERE clause, then (as with UPDATE) the statement affects all rows and you **end up with an empty table!**

  - Well,
    - many database products provide a roll-back mechanism when deleting rows
    - Transactions can also protect you (to some extent)

Delete

- One important point with constraints (foreign keys in particular) is that they guarantee that data remains consistent
  - They don't only work with `insert`, but with `update` and `delete` as well.

  - Example: Try to delete some rows in the `country` table

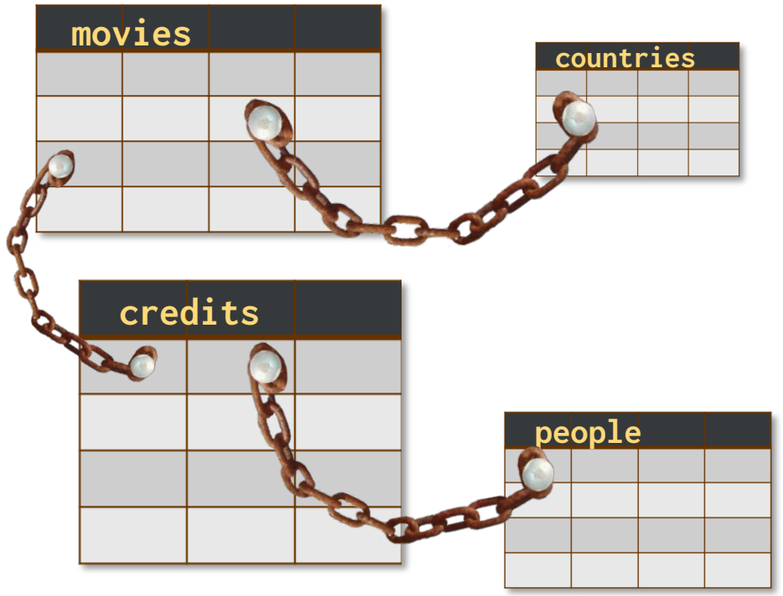
```
delete from countries where country_code = 'us';
```

```
[23503] ERROR: update or delete on table "countries" violates foreign key constraint "movies_country_fkey" on table "movies"
Detail: Key (country_code)=(us) is still referenced from table "movies".
```

    - Foreign-key constraints are especially useful in controlling `delete` operations

Constraints

- This is why constraints are so important:
  - They ensure that whatever happens, you'll always be able to make sense of ALL pieces of data in your database.



Relational Algebra

Relational Algebra

- A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.

- 6 Basic Operators:
  - select: σ
  - project: $\prod$
  - union: $\cup$
  - set difference: $-$
  - Cartesian product: x
  - rename: ρ

Select Operation

- The select operation selects tuples that satisfy a given predicate
  - Notation: $\sigma_p(r)$
  - p is called the selection predicate

- Example
  - Select those tuples of the *instructor* relation where the instructor is in the "Physics" department

$$\sigma_{dept\_name = \text{“}Physics\text{”}}(instructor)$$

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

Select Operation

- We allow comparisons using $=, \neq, >, \geq. <. \leq$ in the selection predicate

- We can combine several predicates into a larger predicate by using the connectives:

$$\wedge \textbf{(and)}, \vee \textbf{(or)}, \neg \textbf{(not)}$$

  - Example: Find the instructors in Physics with a salary greater $90,000, we write:

$$\sigma_{dept\_name = \text{“Physics”} \wedge salary > 90,000}(instructor)$$

- The select predicate may include comparisons between two attributes.
  - Example, find all departments whose name is the same as their building name:

$$\sigma_{dept\_name=building}(department)$$

Project Operation

- A unary operation that returns its argument relation, with certain attributes left out.

Notation: $\prod_{A_1, A_2, A_3 \ldots A_k} (r)$

where $A_1, A_2, \ldots, A_k$ are attribute names and r is a relation name.

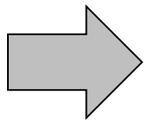
  - The result is defined as the relation of k columns obtained by erasing the columns that are not listed
  - Duplicate rows removed from result, since relations are sets

Project Operation

- Example: eliminate the *dept_name* attribute of instructor
  - Query:

$$\Pi_{ID,\ name,\ salary}\ (instructor)$$

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000



ID	*name*	*salary*
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

Composition of Relational Operations

- The result of a relational-algebra operation is relation
  - … and therefore, relational-algebra operations can be composed together into a relational-algebra expression

- Consider the query: Find the names of all instructors in the Physics department

$$\Pi_{name}(\sigma_{dept\_name = \text{“Physics”}}(instructor))$$

  - Instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation

Cartesian-Product Operation

- The Cartesian-product operation (denoted by ×) allows us to combine information from any two relations.
  - Example: the Cartesian product of the relations instructor and teaches is written as:

$$\text{instructor} \times \text{teaches}$$

- We construct a tuple of the result out of each possible pair of tuples
  - … one from the instructor relation and one from the teaches relation (see next slide)

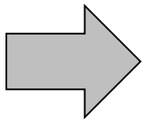
  - Since the instructor ID appears in both relations we distinguish between these attribute by attaching to the attribute the name of the relation from which the attribute originally came.
    - instructor.ID
    - teaches.ID

The "instructor × teaches" table

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

×

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4



instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2017
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2017
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2018
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2017
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2018
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Join Operation

- Problem: The Cartesian-Product "instructor × teaches" associates every tuple of instructor with every tuple of teaches
  - Most of the resulting rows have information about instructors who did NOT teach a particular course

instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...

Join Operation

- To get only those tuples of “instructor × teaches“ that pertain to instructors and the courses that they taught, we write:

$$\sigma_{instructor.id = teaches.id} (instructor \times teaches)$$

  - We get only those tuples of “instructor × teaches” that pertain to instructors and the courses that they taught
    - i.e., those tuples where instructor.id = teaches.id

Join Operation

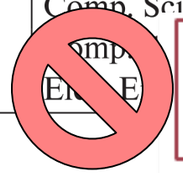
- The table corresponding to $\sigma_{instructor.id = teaches.id}(instructor \times teaches)$:

instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
32343	El Said	History	60000	32343	HIS-351	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-101	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-319	1	Spring	2018
76766	Crick	Biology	72000	76766	BIO-101	1	Summer	2017
76766	Crick	Biology	72000	76766	BIO-301	1	Summer	2018
83821	Brandt	Comp. Sci.	92000	83821	CS-190	1	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-190	2	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-319	2	Spring	2018
98345	Kim	Elec. Eng.	80000	98345	EE-181	1	Spring	2017

Join Operation

- The table corresponding to $\sigma_{instructor.id = teaches.id}(instructor \times teaches)$:

instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
32343	El Said	History	60000	32343	HIS-351	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-101	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-319	1	Spring	2018
76766	Crick	Biology	72000	76766	BIO-101	1	Summer	2017
76766	Crick	Biology	72000	76766	BIO-301	1	Summer	2018
83821	Brandt	Comp. Sci.	92000	83821	CS-190	1	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-190	2	Spring	2017
83821	Brandt	Comp.						
98345	Kim	El	E					



10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...

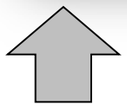
... will NOT include such tuples (rows) with different IDs

Recall: The Old Way of Writing Joins

- Use commas to separate the tables
  - Example: The solution for the same question in the previous slide

- A little bit history:
  - `join` was introduced in SQL-1999 (later than this original way)
- Problem:
  - If you forget a comma, it will still work sometimes (interpreted as “renaming”)

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



The SQL syntax was derived from the form of cartesian products in relational algreba

$$\sigma_{movies.id = credits.id \wedge \text{people.peopleid} = \text{credits.peopleid} \wedge \text{movies.country} = \text{“cn”}} (Movies \times Credits \times People)$$

Recall: The Old Way of Writing Joins

- Use commas to separate the tables
  - Example: The solution for the same question in the previous slide

- A little bit history:
  - `join` was introduced in SQL-1999 (later than this original way)
- Problem:
  - If you forget a comma, it will still work sometimes (interpreted as "renaming")

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

Use commas as the "multiplication signs"

The "select operation" is written as the "where" clause here

The SQL syntax was derived from the form of cartesian products in relational algreba

$$\sigma_{movies.id = credits.id \wedge \text{people.peopleid} = \text{credits.peopleid} \wedge \text{movies.country} = \text{“cn”}} (Movies \times Credits \times People)$$

Join Operation

- The join operation allows us to combine a select operation and a Cartesian-Product operation into a single operation
  - Consider relations $r(R)$ and $s(S)$:
  - Let "theta (θ)" be a predicate on attributes in the schema R "union" S. The join operation $r \bowtie_\theta s$ is defined as follows:

$$r \bowtie_\theta s = \sigma_\theta (r \times s)$$

  - Thus, $\sigma_{instructor.id = teaches.id} (instructor \times teaches)$ can equivalently be written as:

$$instructor \bowtie_{Instructor.id = teaches.id} teaches$$

Union Operation

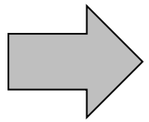
- The union operation allows us to combine two relations
  - Notation: $r \cup s$
- For $r \cup s$ to be valid:
  - r, s must have the same arity (same number of attributes)
  - The attribute domains must be compatible
    - Example: 2nd column of r deals with the same type of values as does the 2nd column of s

Union Operation

- Example: To find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both

$$\Pi_{course\_id}(\sigma_{semester="Fall" \wedge year=2017}(section)) \cup \Pi_{course\_id}(\sigma_{semester="Spring" \wedge year=2018}(section))$$

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A



course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Set-Intersection Operation

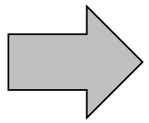
- The set-intersection operation allows us to find tuples that are in both the input relations
  - Notation: $r \cap s$
- Assume (same as Union):
  - r, s have the same arity
  - Attributes of r and s are compatible

Set-Intersection Operation

- Example: Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters

$$\Pi_{course\_id}\,(\sigma_{semester="Fall" \wedge year=2017}(section)) \cap \Pi_{course\_id}\,(\sigma_{semester="Spring" \wedge year=2018}(section))$$

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A



course_id
CS-101

Set Difference Operation

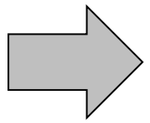
- The set-difference operation allows us to find tuples that are in one relation but are not in another
  - Notation: $r - s$
- Assume (same as Union and Set Intersection):
  - r, s have the same arity
  - Attributes of r and s are compatible

Set Difference Operation

- Example: Find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

$$\Pi_{course\_id}\,(\sigma_{\,semester="Fall" \wedge year=2017}\,(section)) - \Pi_{course\_id}\,(\sigma_{\,semester="Spring" \wedge year=2018}\,(section))$$

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A



course_id
CS-347
PHY-101

The Assignment Operation

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables
  - The assignment operation is denoted by ← and works like assignment in a programming language
- Example: Find all instructor in the "Physics" and Music department

$$Physics \leftarrow \sigma_{dept\_name = \text{“}Physics\text{”}}(instructor)$$
$$Music \leftarrow \sigma_{dept\_name = \text{“}Music\text{”}}(instructor)$$
$$Physics \cup Music$$

  - With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.

The Rename Operation

- The results of relational-algebra expressions do not have a name that we can use to refer to them
  - The rename operator, ρ, is provided for that purpose
  - The expression $\rho_x(E)$ returns the result of expression E under the name x
  - Another form of the rename operation which also renames the columns:
    - $\rho_{x(A1,A2,\ldots An)}(E)$

Equivalent Queries

- There is more than one way to write a query in relational algebra
  - Example: Find information about courses taught by instructors in the Physics department with salary greater than 90,000
    - Query 1

$$\sigma_{dept\_name = \text{“Physics”} \wedge salary > 90{,}000}(instructor)$$

    - Query 2

$$\sigma_{dept\_name = \text{“Physics”}}(\sigma_{salary > 90.000}(instructor))$$

    - The two queries are not identical
      - they are, however, equivalent -- they give the same result on any database

Equivalent Queries

- Example: Find information about courses taught by instructors in the Physics department
  - Query 1

$$\sigma_{dept\_name = \text{“Physics”}} (instructor \bowtie_{instructor.ID = teaches.ID} teaches)$$

    - (Join first, then select)

  - Query 2

$$(\sigma_{dept\_name = \text{“Physics”}} (instructor)) \bowtie_{instructor.ID = teaches.ID} teaches$$

    - (Select first, then join)

Equivalent Queries

- Application of Relational Algebra: Query Optimization
  - Transform queries into equivalent ones with less computational cost