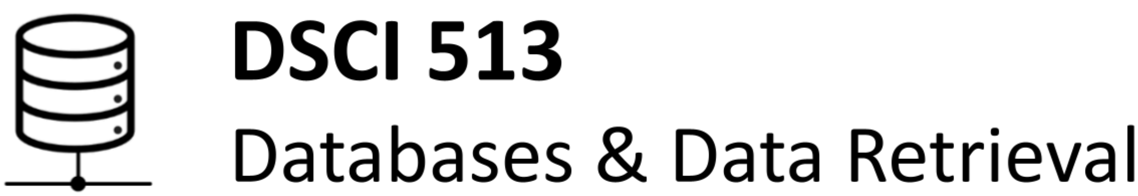
Lecture 1: Introduction to relational databases

Contents



Course announcements

- Significance of this course
- What this course is, and what it isn’t

Skip to main content

Lecture outline

- Why use databases
- The relational model
- Query languages, SQL, Postgres
- How to run SQL
- Basic SQL queries

Why not spreadsheets?

At this point in MDS, you have a good idea of why spreadsheet software like Excel or its equivalents are not suitable for most data science purposes.

Pandas:

- reproducible
- range of functionalities
- scalable
- fast
- can be automated

Why not Pandas dataframes?

But Pandas was pretty nice and powerful, wasn’t it? let’s see.

Question:

What kind of problems do you think you might run into using a pandas dataframe like the above?

Think about what happens if:

- Your dataframe is 100 GB in size

Skip to main content

- You want to be able to manage what each user can do
- You want to be able to let different users see different parts of the dataset
- You don't want to store everything at one place
- You want to restrict the kind of data to be stored
- The dataset file is corrupted
- The system crashes half way through making a change
- You want to optimize access to your data
- ...

Databases and database management systems

A **database management system (DBMS)** addresses all of the above problems.

What is a database? A database is an organized collection of related data

What is a database management system? A DBMS is a collection of programs that enables users to create, query, modify and manage a database in an optimized and efficient manner. A DBMS relieves us from worrying about storing a manage

Using a DBMS ensures:

- Efficient data access
- Data integrity
- Data security
- Concurrent access
- Crash recovery
- Data independence

Remember:

database $\neq$ database management system

Data model

A data model is the way we choose to represent data. We usually try to model the data in a way that is closer to how we think about the data.

You probably remember from when we talked about tidy data, that we like to see

- each observation or measurement as a **row**
- each variable or attribute as **column**

Is that the only way to represent data? No! But that’s the one the makes sense for a variety of applications.

That particular way of representing the data is called a **data model**.

There are different types of DBMS for different kinds of data models:

- Relational (most widely used)
- Document
- Hierarchical
- Network
- Object-oriented
- Graph

Example:

In a graph database for a social media application, people may be represented as the nodes of a graph, whereas graph edges may define the relationship of each person to another person.

Example:

In this course, we'll talk mostly about **relational** DBMSs (RDBMS) and briefly about **non-relational** DBMSs.

The relational model

Why the relational model?

Take a moment and think about the kind of problems that you may run into if you choose to store data in a single table.

	id [PK] integer	**name** text	**email** text	**phone** character varying (12)	**department** character varying (50)
1	4	Varada	varada@mds.ubc.ca	243-924-4446	Computer Science
2	5	Quan	quan@mds.ubc.ca	644-818-0254	Economics
3	6	Joel	joel@mds.ubc.ca	773-432-7669	Biomedical Engineering
4	7	Florencia	flor@mds.ubc.ca	773-926-2837	Biology
5	8	Alexi	alexiu@mds.ubc.ca	421-888-4550	Statistics
6	15	Vincenzo	vincenzo@mds.ubc.ca	776-543-1212	Statistics
7	19	Gittu	gittu@mds.ubc.ca	776-334-1132	Biomedical Engineering
8	16	Jessica	jessica@mds.ubc.ca	211-990-1762	Computer Science
9	1	Tiffany	tiff@mds.ubc.ca	445-794-2233	Neuroscience

The most famous data model today is the relational model, while other models have also gained traction in the past few years.

The relational model works with **entities** and **relationships**. It is based on the set theory in mathematics was introduced by by Edgar Codd (IBM) in 1970 (more details here). It's foundations in **set theory** is the reason you will here words like "tuples", "domain", "union", "cross product", etc.

Example:

Skip to main content

- students in a school
- employees of an organization
- cars of a rental company
- houses in a city

Relations:

- students to a department
- purchases to customers
- movies to actors
- customers to a bank

In a relational model, entities and relationships are both **sets of tuples** called **relations**. These relations are represented as **tables** with rows and columns.

What is a relational database? A collection of relations

A **Relation** is an instance of a schema (just like an object was an instance of a class!)

The **Schema** specifies

1. Name of a relation
2. Name and domain of each attribute

Domain: A set of constraints that determines the type, length, format, range, uniqueness and nullability of values stored for an attribute.

Example:

Student (**sid**: *string*, **name**: *string*, **login**: *string*, **age**: *integer*, **gpa**: *real*)

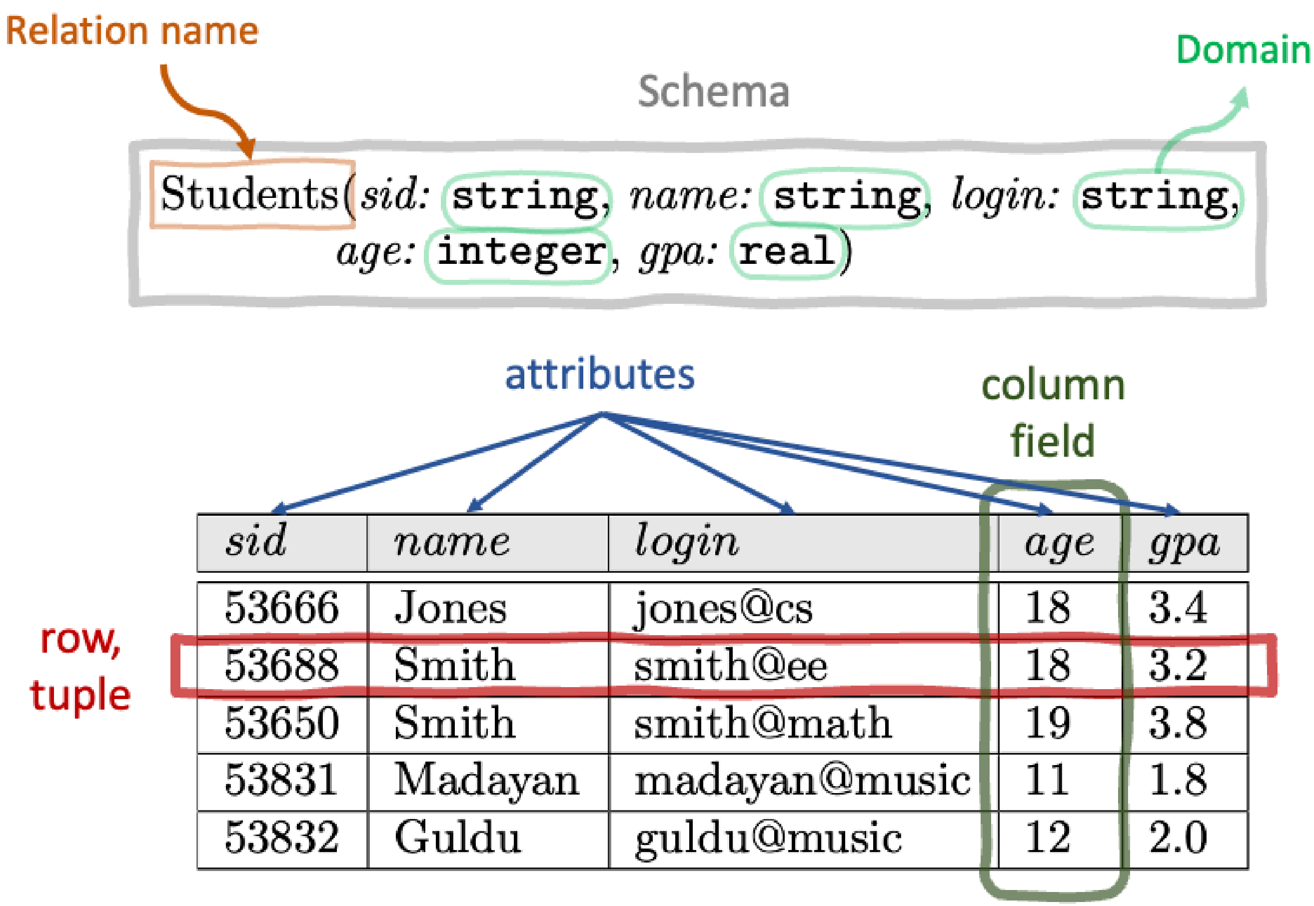
Skip to main content

Instance: a particular relation that follows a certain schema

Relational Database Schema: collection of schemas in the database

Database Instance: a collection of instances of its relations

Anatomy of a table



Question: Can you name a few differences between tables in a database and spreadsheets? How about between tables and Pandas dataframes?

Skip to main content

Properties of a table (or relation)

- Contains data about a particular entity or relationship between entities.
- Has a unique name in a database.
- Each row-column intersection stores a single value, that is to say values should be atomic.
- Has at least one column and zero or more rows.

Properties of rows (or tuples, records)

- The order of rows are not important. Therefore, there is no index-based method to retrieve rows like in Pandas.
- Contains information about an instance of the entity/relationship which the table represents
- Each row is identified by its *primary* key (we'll learn more about this in future lectures)

Properties of columns (or attributes)

- Has a unique name in a table.
- Represents a particular property of the entity/relationship which the table represents
- The order of columns are not important. Therefore, there is no index-based method to retrieve columns like in Pandas.
- Is valued according to a domain (rules for values)

Query language in a DBMS

What is a query? A question that we ask about the data. The result of a query is a new relation.

In order to talk to the database and ask questions, we need to speak its language. A DBMS

- provides a specialized language for us to write our queries
- optimizes how our queries are executed

The data query language (DQL) is part of a bigger set of languages for working with data in a relational database, which consists of

- data definition language (DDL) for creating, altering and deleting tables
- data manipulation language (DML) for inserting new data, updating values, etc.
- data query language (DQL) for querying and retrieving data
- data control language (DCL) for management and controlling user access, rights and privileges

What is SQL?

Well, it's finally time to learn about SQL!

- SQL stands for Structured Query Language (or... does it?).
- It is a programming language that we use to talk to a relational DBMS.
- Originally developed by IBM in 1970s to manipulate and retrieve data stored in their DBMS, System R.
- SQL ≠ relational model ≠ database ≠ DBMS

A peak at SQL queries

Skip to main content

> we want to retrieve the names and GPAs of students older than 25.

sid	name	login	age	gpa
23792	Arman	arman@mds	28	2.5
82347	Varada	varada@mds	29	2.9
11238	Tiffany	tiff@mds	23	2.8
87263	Mike	mike@mds	19	3.8
13298	Joel	joel@mds	25	3.2
91287	Florencia	flor@mds	20	3.3

We can write this as the following SQL query:

```sql
SELECT
    name, age, gpa
FROM
    Students
WHERE
    age > 25;
```

Running the above query should return this relation:

name	age	gpa
Arman	28	2.5
Varada	29	2.9

Skip to main content

SQL syntax

Let’s dissect the different parts of the our SQL query here:

```
SELECT
    name, age, gpa
FROM
    Students
WHERE
    age > 25;
```

A SQL statement consists of keywords, clauses, identifiers, terminating semi-colon and sometimes comments which together form a complete executable and independent piece of code.

```
SELECT
```

- The keyword `SELECT` is the **keyword** that exists in every SQL query. It is used to select and return data from columns, given the conditions that follow it.

```
name, age, gpa
Students
```

- `SELECT` is very powerful, but not dangerous: A `SELECT` statement never changes any values or tables in the database.
- The fact that we select only a few columns (instead of all of them) is called **projection** in database terms.
- These are called **identifiers**, and refer to the labels of columns and tables that exist in the database.

Skip to main content

```
FROM
```

- This is another keyword that tells SQL which relation (i.e. table) to retrieve the columns from.

```
WHERE
```

- Yet another SQL keyword that is used to place a condition (also known as a “predicate”) on the returned values.

- We can also have comments in a SQL query by preceding text with `--`:

```
-- Hey, I'm a comment!
-- ===========================
SELECT
    name, age, gpa  -- column names
FROM
    Students        -- table name
WHERE
    age > 25;       -- condition
```

- Block comments are also possible by enclosing comment lines in `/*` and `*/`:

```
/*
This is our first SQL query, and we
are learning about the following keywords:
SELECT
FROM
WHERE
*/

SELECT
```

Skip to main content

```
    Students
WHERE
    age > 25;
```

- Don't forget that every SQL statement needs to be terminated with a `;`.
- SQL keywords are traditionally written in upper case letters, but that is not a requirement. I prefer to follow this tradition because it makes the query more readable.

- A Keyword, together with its following identifiers, expressions, etc. is collectively called a **clause**. For example:

```
SELECT
    name, age, gpa  -- columns are chosen here
FROM
    Students        -- table is specified here
WHERE
    age > 25;       -- filter is applied here
```

- It is common to put each clause or each keyword on a different line, but there is no generally agreed-upon style.
- In general, it doesn't matter whether the entire SQL statement is on one line or broken over several lines. Anything that comes before a `;` belongs to the same statement.

There are many other keywords that we will use throughout DSCI 513. The ones that you just saw are a few that are usually used when querying data.

> Note that SQL **is not imperative** (like Python or C++); it is a **declarative** language: We don't tell SQL **how** to retrieve data, but **what** to retrieve.

Skip to main content

> For instance, we didn't write a for loop to retrieve the data from each row according to a certain condition. We told SQL what we wanted, and SQL did it for us.

Flavours of SQL

- SQL is not owned by a particular company or organization
- It became a database language standard by the American National Standards Institute (ANSI) in 1986, and the International Organization for Standardization (ISO) in 1987.
- However, there are various SQL flavors and implementations, such as Oracle SQL, MySQL (open source), PostgreSQL (open source), IBM DB2, Microsoft SQL Server, Microsoft Access, SQLite (open source)
- These implementations have slightly different syntax and various additional features.
- In DSCI 513, we use **PostgreSQL**

What is PostgreSQL?

PostgreSQL (also known simply by its nickname *Postgres*) is an open-source, cross-platform DBMS that implements the relational model. PostgreSQL is very reliable with great performance characteristics, and is equipped with almost all features of the commercial and proprietary DBMSs.

PostgreSQL appeared in 1980s as a research project in University of California, Berkeley. It was meant to improve an earlier prototype relational DBMS called INGRES, which explains the name Postgres, which is short for PostINGRES. Here is an informative blog post about PostgreSQL's history if you're interested!

How to run SQL in Postgres?

Well, we have a variety of options to run our SQL statements in PostgreSQL, the most common of which are the following:

- pgAdmin is the official web-based GUI for interacting with PostgreSQL databases
- `psql` is PostgreSQL's interactive command-line interface
- `%sql` and `%%sql` magic commands in Jupyter notebooks, which are provided by the `ipython-sql` package
- `psycopg2` is the official Python adapter for PostgreSQL databases
- Using `.read_sql_query()` method in Pandas

I will demonstrate the usage of all these interfaces here.

pgAdmin

I will demo this in the lecture.

> I like to use a `Shift + Enter` keyboard shortcut to run my queries. You can configure this too by going Preferences -> Query Tool -> Keyboard shortcuts -> Change "Execute query" to `Shift + Enter`.

`psql`

- SQL statements, as well as
- `psql`'s special "meta" commands.

I introduce a couple of useful `psql` meta commands here, but you can find all the other ones in Postgres documentations here or a shorter version in this cheatsheet.

Command	Usage
`\l`	list all databases
`\c`	connect to a database
`\d`	describe tables and views
`\dt`	list tables
`\dt+`	list tables with additional info
`\d+`	list tables and views with additional info
`\!`	execute shell commands
`\cd`	change directory
`\i`	execute commands from file
`\h`	view help on SQL commands
`\?`	view help on psql meta commands
`\q`	quit interactive shell

> Note that you don't need to terminate meta commands with `;`.

`ipython-sql` (`%sql` and `%%sql` magics)

`ipython-sql` is a package that enables us to run SQL statements right from a Jupyter notebook. This package is included in the `dsci513env.yaml` environment file, so you

```
%load_ext sql
```

Now we need the host address of where the database is stored, along with a username and a password.

It is always a bad idea to store login information directly in a notebook or code file because of security reasons. For example, you don't want to commit your sensitive login information to a Git repo.

In order to avoid that, we store that kind of information in a separate file, like `credentials.json` here, and read the username and password into our IPython session:

```
import json
import urllib.parse

with open('data/credentials.json') as f:
    login = json.load(f)

username = login['user']
password = urllib.parse.quote(login['password'])
host = login['host']
port = login['port']
```

And also make sure to add your file name (e.g. `credentials.json`) to your `.gitignore` file, so you don't accidentally commit it.

Now we can establish the connection to the `world` database using the following code:

```
%sql postgresql://{username}:{password}@{host}:{port}/world
```

Note that we have used the `%sql` line magic to interpret the line in front of it as a magic command. This is similar to the `%timit` magic that we used in DSCI 511 and 512.

We can also use `%%sql` cell magic to apply the magic to an entire notebook cell.

Skip to main content

```
%sql SELECT name, population FROM country;
```

```
 * postgresql://postgres:***@localhost:5432/world
239 rows affected.
```

name	population
Afghanistan	22720000
Netherlands	15864000
Netherlands Antilles	217000
Albania	3401200
Algeria	31471000
American Samoa	68000
Andorra	78000
Angola	12878000
Anguilla	8000
Antigua and Barbuda	68000
United Arab Emirates	2441000
Argentina	37032000
Armenia	3520000
Aruba	103000
Australia	18886000
Azerbaijan	7734000
Bahamas	307000
Bahrain	617000
Bangladesh	129155000
Barbados	270000
Belgium	10239000
Belize	241000
Benin	6097000
Bermuda	65000
Bhutan	2124000
Bolivia	8329000
Bosnia and Herzegovina	3972000

Skip to main content

Botswana	1622000
Brazil	170115000
United Kingdom	59623400
Virgin Islands, British	21000
Brunei	328000
Bulgaria	8190900
Burkina Faso	11937000
Burundi	6695000
Cayman Islands	38000
Chile	15211000
Cook Islands	20000
Costa Rica	4023000
Djibouti	638000
Dominica	71000
Dominican Republic	8495000
Ecuador	12646000
Egypt	68470000
El Salvador	6276000
Eritrea	3850000
Spain	39441700
South Africa	40377000
Ethiopia	62565000
Falkland Islands	2000
Fiji Islands	817000
Philippines	75967000
Faroe Islands	43000
Gabon	1226000
Gambia	1305000

Skip to main content

Georgia	4968000
Ghana	20212000
Gibraltar	25000
Grenada	94000
Greenland	56000
Guadeloupe	456000
Guam	168000
Guatemala	11385000
Guinea	7430000
Guinea-Bissau	1213000
Guyana	861000
Haiti	8222000
Honduras	6485000
Hong Kong	6782000
Svalbard and Jan Mayen	3200
Indonesia	212107000
India	1013662000
Iraq	23115000
Iran	67702000
Ireland	3775100
Iceland	279000
Israel	6217000
Italy	57680000
East Timor	885000
Austria	8091800
Jamaica	2583000
Japan	126714000
Yemen	18112000

Skip to main content

Jordan	5083000
Christmas Island	2500
Yugoslavia	10640000
Cambodia	11168000
Cameroon	15085000
Canada	31147000
Cape Verde	428000
Kazakstan	16223000
Kenya	30080000
Central African Republic	3615000
China	1277558000
Kyrgyzstan	4699000
Kiribati	83000
Colombia	42321000
Comoros	578000
Congo	2943000
Congo, The Democratic Republic of the	51654000
Cocos (Keeling) Islands	600
North Korea	24039000
South Korea	46844000
Greece	10545700
Croatia	4473000
Cuba	11201000
Kuwait	1972000
Cyprus	754700
Laos	5433000
Latvia	2424200
Lesotho	2153000

Skip to main content

Lebanon	3282000
Liberia	3154000
Libyan Arab Jamahiriya	5605000
Liechtenstein	32300
Lithuania	3698500
Luxembourg	435700
Western Sahara	293000
Macao	473000
Madagascar	15942000
Macedonia	2024000
Malawi	10925000
Maldives	286000
Malaysia	22244000
Mali	11234000
Malta	380200
Morocco	28351000
Marshall Islands	64000
Martinique	395000
Mauritania	2670000
Mauritius	1158000
Mayotte	149000
Mexico	98881000
Micronesia, Federated States of	119000
Moldova	4380000
Monaco	34000
Mongolia	2662000
Montserrat	11000
Mozambique	19680000

Skip to main content

Myanmar	45611000
Namibia	1726000
Nauru	12000
Nepal	23930000
Nicaragua	5074000
Niger	10730000
Nigeria	111506000
Niue	2000
Norfolk Island	2000
Norway	4478500
Côte d Ivoire	14786000
Oman	2542000
Pakistan	156483000
Palau	19000
Panama	2856000
Papua New Guinea	4807000
Paraguay	5496000
Peru	25662000
Pitcairn	50
Northern Mariana Islands	78000
Portugal	9997600
Puerto Rico	3869000
Poland	38653600
Equatorial Guinea	453000
Qatar	599000
France	59225700
French Guiana	181000
French Polynesia	235000

Skip to main content

Réunion	699000
Romania	22455500
Rwanda	7733000
Sweden	8861400
Saint Helena	6000
Saint Kitts and Nevis	38000
Saint Lucia	154000
Saint Vincent and the Grenadines	114000
Saint Pierre and Miquelon	7000
Germany	82164700
Solomon Islands	444000
Zambia	9169000
Samoa	180000
San Marino	27000
Sao Tome and Principe	147000
Saudi Arabia	21607000
Senegal	9481000
Seychelles	77000
Sierra Leone	4854000
Singapore	3567000
Slovakia	5398700
Slovenia	1987800
Somalia	10097000
Sri Lanka	18827000
Sudan	29490000
Finland	5171300
Suriname	417000
Swaziland	1008000

Skip to main content

Switzerland	7160400
Syria	16125000
Tajikistan	6188000
Taiwan	22256000
Tanzania	33517000
Denmark	5330000
Thailand	61399000
Togo	4629000
Tokelau	2000
Tonga	99000
Trinidad and Tobago	1295000
Chad	7651000
Czech Republic	10278100
Tunisia	9586000
Turkey	66591000
Turkmenistan	4459000
Turks and Caicos Islands	17000
Tuvalu	12000
Uganda	21778000
Ukraine	50456000
Hungary	10043200
Uruguay	3337000
New Caledonia	214000
New Zealand	3862000
Uzbekistan	24318000
Belarus	10236000
Wallis and Futuna	15000
Vanuatu	190000

Skip to main content

Holy See (Vatican City State)	1000
Venezuela	24170000
Russian Federation	146934000
Vietnam	79832000
Estonia	1439200
United States	278357000
Virgin Islands, U.S.	93000
Zimbabwe	11669000
Palestine	3101000
Antarctica	0
Bouvet Island	0
British Indian Ocean Territory	0
South Georgia and the South Sandwich Islands	0
Heard Island and McDonald Islands	0
French Southern territories	0
United States Minor Outlying Islands	0

Limiting returned and displayed rows

As you can see, all rows are returned and displayed by default. This behaviour can be problematic if our table is very large for two reasons:

1. Retrieving large tables can be slow, and maybe not necessary
2. Displaying a lot of rows clutters our Jupyter notebook

We can modifying `ipython-sql` configuration to limit the number of returned and displayed rows. For example, here we change the display limit:

Skip to main content

```
%config SqlMagic.displaylimit = 20
```

```
%sql SELECT name, population FROM country;
```

```
 * postgresql://postgres:***@localhost:5432/world
239 rows affected.
```

name	population
Afghanistan	22720000
Netherlands	15864000
Netherlands Antilles	217000
Albania	3401200
Algeria	31471000
American Samoa	68000
Andorra	78000
Angola	12878000
Anguilla	8000
Antigua and Barbuda	68000
United Arab Emirates	2441000
Argentina	37032000
Armenia	3520000
Aruba	103000
Australia	18886000
Azerbaijan	7734000
Bahamas	307000
Bahrain	617000
Bangladesh	129155000
Barbados	270000

239 rows, truncated to displaylimit of 20

Skip to main content

Looks good. Let’s apply the magic to an entire cell so that we can break the lines:

```
%%sql

SELECT
    name, population
FROM
    country
;
```

```
 * postgresql://postgres:***@localhost:5432/world
239 rows affected.
```

name	population
Afghanistan	22720000
Netherlands	15864000
Netherlands Antilles	217000
Albania	3401200
Algeria	31471000
American Samoa	68000
Andorra	78000
Angola	12878000
Anguilla	8000
Antigua and Barbuda	68000
United Arab Emirates	2441000
Argentina	37032000
Armenia	3520000
Aruba	103000
Australia	18886000
Azerbaijan	7734000
Bahamas	307000
Bahrain	617000
Bangladesh	129155000
Barbados	270000

239 rows, truncated to displaylimit of 20

We can use `*` to retrieve all columns:

```
%%sql
```

Skip to main content

```
    *
FROM
    country
;
```

```
 * postgresql://postgres:***@localhost:5432/world
239 rows affected.
```

Skip to main content

code	name	continent	region	surfacearea	indepyear	population	lifee
AFG	Afghanistan	Asia	Southern and Central Asia	652090.0	1919	22720000	
NLD	Netherlands	Europe	Western Europe	41526.0	1581	15864000	
ANT	Netherlands Antilles	North America	Caribbean	800.0	None	217000	
ALB	Albania	Europe	Southern Europe	28748.0	1912	3401200	
DZA	Algeria	Africa	Northern Africa	2381741.0	1962	31471000	
ASM	American Samoa	Oceania	Polynesia	199.0	None	68000	
AND	Andorra	Europe	Southern Europe	468.0	1278	78000	
AGO	Angola	Africa	Central Africa	1246700.0	1975	12878000	
AIA	Anguilla	North America	Caribbean	96.0	None	8000	
ATG	Antigua and Barbuda	North America	Caribbean	442.0	1981	68000	
ARE	United Arab Emirates	Asia	Middle East	83600.0	1971	2441000	
ARG	Argentina	South America	South America	2780400.0	1816	37032000	
ARM	Armenia	Asia	Middle East	29800.0	1991	3520000	
ABW	Aruba	North America	Caribbean	193.0	None	103000	
AUS	Australia	Oceania	Australia and New	7741220.0	1901	18886000	

Skip to main content

AZE	Azerbaijan	Asia	Middle East	86600.0	1991	7734000
BHS	Bahamas	North America	Caribbean	13878.0	1973	307000
BHR	Bahrain	Asia	Middle East	694.0	1971	617000
BGD	Bangladesh	Asia	Southern and Central Asia	143998.0	1971	129155000
BRB	Barbados	North America	Caribbean	430.0	1966	270000

239 rows, truncated to displaylimit of 20

Assigning returned rows to Python variables (time permitting)

Single line queries:

```
query_output = %sql SELECT name, population FROM country;
```

```
 * postgresql://postgres:***@localhost:5432/world
239 rows affected.
```

Multi-line queries:

```
%%sql query_output <<

SELECT
    name, population
FROM
    country
;
```

Skip to main content

```
Returning data to local variable query_output
```

```
query_output
```

name	population
Afghanistan	22720000
Netherlands	15864000
Netherlands Antilles	217000
Albania	3401200
Algeria	31471000
American Samoa	68000
Andorra	78000
Angola	12878000
Anguilla	8000
Antigua and Barbuda	68000
United Arab Emirates	2441000
Argentina	37032000
Armenia	3520000
Aruba	103000
Australia	18886000
Azerbaijan	7734000
Bahamas	307000
Bahrain	617000
Bangladesh	129155000
Barbados	270000

239 rows, truncated to displaylimit of 20

Although this looks like a dataframe, `query_output` is not a dataframe:

Skip to main content

```
sql.run.ResultSet
```

But we can easily convert it to a Pandas dataframe:

```
df = query_output.DataFrame()
```

```
type(df)
```

```
pandas.core.frame.DataFrame
```

```
df
```

	name	population
0	Afghanistan	22720000
1	Netherlands	15864000
2	Netherlands Antilles	217000
3	Albania	3401200
4	Algeria	31471000
...	...	...
234	British Indian Ocean Territory	0
235	South Georgia and the South Sandwich Islands	0
236	Heard Island and McDonald Islands	0
237	French Southern territories	0
238	United States Minor Outlying Islands	0

239 rows × 2 columns

```
df.loc[df['name'] == 'Canada', 'population']
```

Skip to main content

```
Name: population, dtype: int64
```

If you want the result of every query to be automatically converted to a Pandas dataframe, there is an option for that in `ipython-sql`:

```
%config SqlMagic.autopandas = True
```

```
new_query = %sql SELECT name, population FROM country;
new_query
```

```
 * postgresql://postgres:***@localhost:5432/world
239 rows affected.
```

	name	population
0	Afghanistan	22720000
1	Netherlands	15864000
2	Netherlands Antilles	217000
3	Albania	3401200
4	Algeria	31471000
...	...	...
234	British Indian Ocean Territory	0
235	South Georgia and the South Sandwich Islands	0
236	Heard Island and McDonald Islands	0
237	French Southern territories	0
238	United States Minor Outlying Islands	0

239 rows × 2 columns

```
type(new_query)
```

```
pandas.core.frame.DataFrame
```

Skip to main content

```
%config SqlMagic.autopandas = False
```

Embedding variables

Much like when we embed variables in strings in Python using f-strings, we can do the same in `ipython-sql` by preceding the variable name with a `:`, i.e. a colon:

```
loc1 = 'Canada'

%sql SELECT name, population FROM country WHERE name = :loc1
```

```
 * postgresql://postgres:***@localhost:5432/world
1 rows affected.
```

name	population
Canada	31147000

More SQL commands

`DISTINCT`

The `DISTINCT` keyword is used to return only distinct rows from a table, and ignore duplicates:

```
SELECT
    DISTINCT column1, column2, ...
```

Skip to main content

```
FROM
    table1;
```

Note that `DISTINCT` is applied to **all columns that we list in front of `SELECT`**, and returns all distinct combinations of values stored in those columns. In the above code snippet, columns other than `column1` and `column2` can still have duplicate values.

```
%%sql

SELECT
    DISTINCT continent
FROM
    country
;
```

```
 * postgresql://postgres:***@localhost:5432/world
7 rows affected.
```

continent
Asia
South America
North America
Oceania
Antarctica
Africa
Europe

```
%%sql

SELECT
    DISTINCT continent, region
FROM
    country
;
```

Skip to main content

```
 * postgresql://postgres:***@localhost:5432/world
25 rows affected.
```

continent	region
Oceania	Melanesia
Oceania	Australia and New Zealand
North America	Central America
Africa	Northern Africa
Asia	Eastern Asia
Oceania	Polynesia
Europe	Nordic Countries
Asia	Middle East
Oceania	Micronesia/Caribbean
Europe	Baltic Countries
Asia	Southern and Central Asia
Africa	Southern Africa
Africa	Western Africa
Oceania	Micronesia
Europe	British Islands
North America	Caribbean
Asia	Southeast Asia
Africa	Central Africa
Europe	Western Europe
Europe	Southern Europe

25 rows, truncated to displaylimit of 20

Skip to main content

`ORDER BY`

The `ORDER BY` keyword sorts the results according to one or particular set of columns:

```
SELECT
    column1, column2, ...
FROM
    table1
ORDER BY
    column1, column2, ...;
```

The rows are sorted in **ascending** order by default.

```
%%sql

SELECT
    name, population
FROM
    country
ORDER
    BY population
;
```

```
 * postgresql://postgres:***@localhost:5432/world
239 rows affected.
```

name	population
Heard Island and McDonald Islands	0
United States Minor Outlying Islands	0
South Georgia and the South Sandwich Islands	0
Antarctica	0
Bouvet Island	0
British Indian Ocean Territory	0
French Southern territories	0
Pitcairn	50
Cocos (Keeling) Islands	600
Holy See (Vatican City State)	1000
Niue	2000
Tokelau	2000
Falkland Islands	2000
Norfolk Island	2000
Christmas Island	2500
Svalbard and Jan Mayen	3200
Saint Helena	6000
Saint Pierre and Miquelon	7000
Anguilla	8000
Montserrat	11000

239 rows, truncated to displaylimit of 20

We can also sort the returned rows in descending order by adding the keyword `DESC` keyword after the column names. In fact, there is a `ASC` keyword as well for ascending sorting, which is optional:

```
SELECT
    column1, column2, ...
FROM
    table1
```

```
ORDER BY
    column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

```
%%sql

SELECT
    name, population
FROM
    country
ORDER BY
    population DESC
;
```

```
 * postgresql://postgres:***@localhost:5432/world
239 rows affected.
```

name	population
China	1277558000
India	1013662000
United States	278357000
Indonesia	212107000
Brazil	170115000
Pakistan	156483000
Russian Federation	146934000
Bangladesh	129155000
Japan	126714000
Nigeria	111506000
Mexico	98881000
Germany	82164700
Vietnam	79832000
Philippines	75967000
Egypt	68470000
Iran	67702000
Turkey	66591000
Ethiopia	62565000
Thailand	61399000
United Kingdom	59623400

239 rows, truncated to displaylimit of 20

`LIMIT`

Skip to main content

in general, we can use the `LIMIT` keyword to limit the number of returned rows:

```
SELECT
    column1, column2, ...
FROM
    table1
LIMIT
    N_ROWS;
```

```
%%sql

SELECT
    name, continent
FROM
    country
LIMIT
    5
;
```

```
 * postgresql://postgres:***@localhost:5432/world
5 rows affected.
```

name	continent
Afghanistan	Asia
Netherlands	Europe
Netherlands Antilles	North America
Albania	Europe
Algeria	Africa

It is also possible to skip the first `n` rows by supplying the optional `OFFSET` keyword:

```
SELECT
    column1, column2, ...
FROM
    table1
LIMIT
    N_ROWS OFFSET N_OFFSET;
```

```
%%sql
```

Skip to main content

```
    name, continent
FROM
    country
LIMIT
    5 OFFSET 10;
```

```
 * postgresql://postgres:***@localhost:5432/world
5 rows affected.
```

name	continent
United Arab Emirates	Asia
Argentina	South America
Armenia	Asia
Aruba	North America
Australia	Oceania

The order of SQL clauses does matter. The correct order for the ones that we've learned so far is:

- `SELECT`
- `FROM`
- `WHERE`
- `ORDER BY`
- `LIMIT`

More on this in the next lectures.

(optional) SQL-Pandas similarity

SQL Command	Functionality	Example	Pandas Equivalent
`SELECT`	Extracts data from a database	`SELECT surname, email FROM info;`	`df[["surname", "email"]]`
`LIMIT`	Limits the number of rows returned	`SELECT * FROM info` `LIMIT 5;`	`df.head()`
`COUNT()`	Counts how many rows returned, note the parentheses because it's a function	`SELECT COUNT(*) FROM info;`	`df.shape[0]`
`SELECT DISTINCT`	Returns only unique values	`SELECT DISTINCT city FROM info;`	`df.drop_duplicates()`
`WHERE`	Filters data based on a condition(s) like `>`, `<`, `=`, `!=`, etc.)	`SELECT * FROM info` `WHERE city='Vancouver';`	`df.query("city == 'Vancouver'")`
`ORDER BY`	Sorts returned data in ascending (default) or descending order	`SELECT * FROM info` `ORDER BY stu_id` `SELECT * FROM info` `ORDER BY stu_id DESC`	`df.sort_values(by="stu_id")`
`MIN()`, `MAX()`, `AVG()`	Performs specified operation on	`SELECT MIN(dsci_511) FROM grades`	`df["dsci_511"].min()`

Skip to main content

More resources on comparison of SQL and Pandas for data retrieval

- Pandas documentation
- Pandas cheatsheet for SQL people from Kaggle