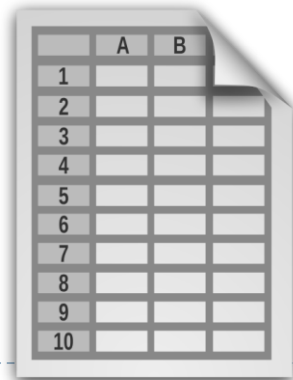# Database and Data Wrangling

Szu-Chi Chung

Department of Applied Mathematics, National Sun Yat-sen University

Taxonomy of the data

▸ Data is facts, information, observations and measurements that are used to make discoveries and to support informed decisions. A data point is a single unit of data with in a dataset, which is collection of data point

▸ Datasets may come in different formats and structures, and will usually be classified based on its source, or where the data came from

> ▸ For example, a company's monthly earnings might be in a spreadsheet but hourly heart rate data from a smartwatch or simple personal information in a club may be in JSON format. It's common for data scientists to work with different types of data within a dataset


```
JSON

{
    "firstName": "John",
    "lastName": "Doe",
    "age": "23",
    "hobbies": [
        { "type": "Sports", "value": "Golf" },
        { "type": "Leisure", "value": "Reading" },
        { "type": "Leisure", "value": "Guitar" }
    ]
}
```

https://docs.microsoft.com/en-us/learn/modules/choose-storage-approach-in-azure/2-classify-data

Database and data Wrangling

- The data you need will often live in databases, systems designed for efficiently storing and querying data. *You will need to know how to read it and extract the information you want*
  - The bulk of these are relational databases, such as SQLite, MySQL, and SQL Server, which store data in tables and are typically queried using Structured Query Language (SQL), a declarative language for manipulating data
- While databases offer very efficient ways to store data and query them using query languages, the most flexible way of data processing is writing your own program to manipulate data
  - In some cases when more complex data processing is needed, it cannot be done easily using SQL
  - Data wrangling using Dataframe is a rescue

1. How data is Described?

▸ **Raw data** is data that has come from its source in its initial state and has not been analyzed or organized

  ▸ In order to make sense of what is happening with a dataset, it needs to be organized into a format that can be understood by humans or can be analyze further

▸ The structure of a dataset describes how it's organized and can be classified at *structured, unstructured and semi-structured*

  ▸ Three common options for storage are text files, spreadsheets, and databases.

  ▸ Text files are easiest to create, and work well with version control, but then we would have to build search and analysis tools ourselves

  ▸ Spreadsheets are good for analysis, but they don't handle large or complex datasets well

  ▸ Relational databases, however, include powerful tools for search and simple analysis, and can handle large, complex datasets

1. How data is Described?

▸ **Unstructured data** typically cannot be categorized into rows or columns and doesn't contain a format or set of rules to follow

  ▸ Because unstructured data has less restrictions on its structure it's easier to add new information in comparison to a structured dataset

  ▸ If a sensor capturing data on barometric pressure every 2 minutes has received an update that now allows it to measure and record temperature, it doesn't require altering the existing data if it's unstructured. However, this may make analyzing or investigating this type of data take longer

  ▸ Examples of unstructured data

    ▸ Media files, such as photos, videos, and audio files

    ▸ Office files, such as Word documents

    ▸ Text files

    ▸ Log files

1. How data is Described?

▸ **Semi-structured data** has features that make it a combination of structured and unstructured data and is not stored in a relational format

    ▸ It doesn't typically conform to a format of rows and columns but is organized in a way that is considered structured and may follow a fixed format or set of rules. The structure will vary between sources, such as a well defined hierarchy to something more flexible that allows for easy integration of new information

    ▸ Metadata are used as indicators that help decide how the data is organized and stored. Some common names for metadata are tags, elements, entities and attributes. For example, a typical email message will have a subject, body and a set of recipients and can be organized by whom or when it was sent

    ▸ Examples of semi-structured data: HTML, CSV files, JavaScript Object Notation (JSON) or NoSQL

1. How data is Described?

▸ **Structured data** is data that is organized into rows and columns, where each row will have the same set of columns and it adheres to a strict schema

▸ Columns will often have a *specific set of rules or restrictions on the values,* to ensure that the values accurately represent the column

▸ A benefit is that it can be organized in such a way that it can be *related* to other structured data. However, making changes to its overall structure can take a lot of effort to do. For example, adding an email column to the customer spreadsheet that cannot be empty is tedious

▸ Examples of structured data: relational databases

id	name	age
1	Jim	28
2	Pam	26
3	Michael	42

id	subject	Teacher
1	Languages	John Jones
2	Track	Wally West
3	Swimming	Arthur Curry
4	Computers	Victor Stone

| student_id	subject_id	grade
2	1	98
1	2	100
1	4	75
3	3	60
2	4	76
3	2	88

2. Relational Database

‣ Recall that the data scientist is someone who cleans, massages, and organizes (big) data. Data scientists will usually already get data that has *passed a first round of cleaning and manipulation*

  ‣ Chances are you have used a spreadsheet in the past to store information. You had a set of rows and columns, where the rows contained the information (or data), and the columns described the information (sometimes called metadata)

  ‣ A relational database is built upon this core principle of columns and rows in tables, allowing you to have information spread across multiple tables. This allows you to work with more complex data, avoid duplication, and have flexibility

  ‣ A DataBase Management System **(DBMS)** is software that allows applications to store and analyze information in a database

    ‣ ACID (atomicity, consistency, isolation, durability) is a set of properties of database transactions intended to guarantee data validity despite errors, power failures, and other mishaps

Data Model

- Proposed in 1970 by Ted Codd

- Database abstraction
  - Store database in simple data structures.
  - Access data through high-level language.
  - Physical storage left up to implementation

- A data model is collection of concepts for describing the data in a database
  - **Relational**
  - Key/Value
  - Graph
  - Document
  - Column-family
  - Array/Matrix

Relational model

▸ A relation is unordered set that contain the relationship of attributes that represent entities

  ▸ A tuple is a set of attribute values in the relation

  ▸ n-ary Relation = Table with n columns

  ▸ A relation's *primary key* uniquely identifies a single tuple and is the domain of n-ary relations

  ▸ A *foreign key* specifies that an attribute from one relation has to map to a tuple in another relation
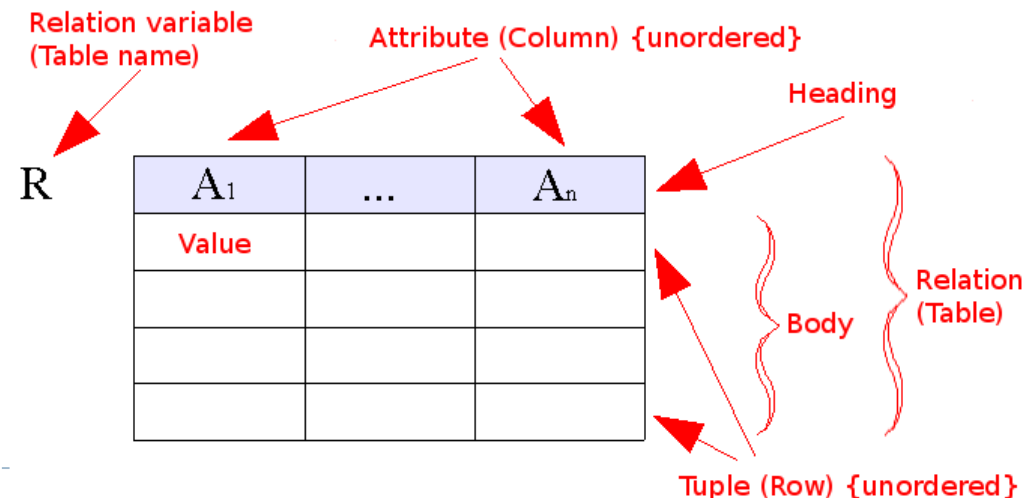
Relational Algebra

▸ Fundamental operations to retrieve and manipulate tuples in a relation.

  ▸ Based on set algebra

  ▸ Each operator takes one or more relations as its inputs and outputs a new relation

  ▸ We can "chain" operators together to create more complex operations

▸ Relation algebra are implemented and extended as queries which are written in SQL. SQL provides hundreds of different ways to analyze and recombine data. We will only look at a handful of queries

  ▸ When we are using a spreadsheet, we put formulas into cells to calculate new values based on old ones. When we are using a database, we send commands (usually called queries) to a database manager. The database manager does whatever lookups and calculations the query specifies, returning the results in a tabular form that we can then use as a starting point for further queries

Schema

▸ The structure of a table is called its **schema**. We need to understand a table's schema to effectively pull out the data we want

> ▸ The database schema is its structure described in a formal language supported by the DBMS. The term "schema" refers to the organization of data as a blueprint of how the database is constructed (divided into database tables in the case of relational databases)

`pets_and_toys` table schema

```
SchemaField('ID', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Name', 'STRING', 'NULLABLE', None, ()),
SchemaField('Age', 'INTEGER', 'NULLABLE', None, ()),
SchemaField('Animal', 'STRING', 'NULLABLE', None, ()),
SchemaField('Toy', 'RECORD', 'NULLABLE', None, (
    SchemaField('Name', 'STRING', 'NULLABLE', None, ()),
    SchemaField('Type', 'STRING', 'NULLABLE', None, ())
))
```

Foundation of extraction – SELECT, FROM, WHERE

▶ The most basic SQL query selects a single column from a single table.

   ▶ specify the column you want after the word SELECT, and then

   ▶ specify the table after the word FROM

▶ For instance, to select the `Name` column (from the `pets` table in the `pet_records` database), our query would appear as follows:

```
query = """
        SELECT Name
        FROM `bigquery-public-data.pet_records.pets`
        """
```

ID	Name	Animal
1	Dr. Harris Bonkers	Rabbit
2	Moon	Dog
3	Ripley	Cat
4	Tom	Cat

https://www.kaggle.com/learn/intro-to-sql

Foundation of extraction – SELECT, FROM, WHERE

▶ Datasets are usually very large, so you'll usually want to return only the rows meeting specific conditions. You can do this using the WHERE clause

> ▶ The query below returns the entries from the `Name` column that are in rows where the `Animal` column has the text `'Cat'`

```
query = """
        SELECT Name
        FROM `bigquery-public-data.pet_records.pets`
        WHERE Animal = 'Cat'
        """
```

ID	Name	Animal
1	Dr. Harris Bonkers	Rabbit
2	Moon	Dog
3	Ripley	Cat
4	Tom	Cat

Get more interesting insights – GROUP BY, HAVING, COUNT

‣ This can help you answer questions like:

  ‣ How many of each kind of fruit has our store sold?

  ‣ How many species of animal has the vet office treated?

‣ COUNT() returns a count of things. If you pass it the name of a column, it will return the number of entries in that column

  ‣ For instance, if we SELECT the COUNT() of the `ID` column in the `pets` table, it will return 4, because there are 4 ID's in the table

  ‣ COUNT() is an example of an **aggregate function**, which takes many values and returns one. Other examples of aggregate functions include SUM(), AVG(), MIN(), and MAX()

```
query = """"""
    SELECT COUNT(ID)
    FROM `bigquery-public-data.pet_records.pets`
    """""""
```

f0_
4

Get more interesting insights – GROUP BY, HAVING, COUNT

▶ GROUP BY takes the name of one or more columns, and treats all rows with the same value in that column as a single group when you apply aggregate functions like COUNT()

   ▶ We can use GROUP BY to group together rows that have the same value in the `animal` column, while using COUNT() to find out how many ID's we have in each group If we want to know how many of each type of animal in the `pets` table

```
query = """
        SELECT Animal, COUNT(ID)
        FROM `bigquery-public-data.pet_records.pets`
        GROUP BY Animal
        """
```

Animal	f0_
Rabbit	1
Dog	1
Cat	2

Get more interesting insights – GROUP BY, HAVING, COUNT

▶ HAVING is used in combination with GROUP BY to ignore groups that don't meet certain criteria

  ▶ So this query, for example, will only include groups that have more than one ID in them

  ▶ HAVING clause behaves similarly to a WHERE clause, except that its filter is applied to the aggregates (whereas a WHERE would filter out rows before aggregation even took place)

```
query = """
        SELECT Animal, COUNT(ID)
        FROM `bigquery-public-data.pet_records.pets`
        GROUP BY Animal
        HAVING COUNT(ID) > 1
        """
```

Animal	f0_
Cat	2

Order results to focus on the most important data – ORDER BY

▶ Now you'll learn how to change the order of your results using the ORDER BY clause, and you'll explore a popular use case by applying ordering to dates

  ▶ ORDER BY is usually the **last clause** in your query, and it sorts the results returned by the rest of your query

  ▶ Notice that the rows are not ordered by the ID column. We can quickly remedy this with the query below

ID	Name	Animal
1	Dr. Harris Bonkers	Rabbit
4	Tom	Cat
2	Moon	Dog
3	Ripley	Cat

```
query = """
        SELECT ID, Name, Animal
        FROM `bigquery-public-data.pet_records.pets`
        ORDER BY ID
        """
```

ID	Name	Animal
1	Dr. Harris Bonkers	Rabbit
2	Moon	Dog
3	Ripley	Cat
4	Tom	Cat

Order results to focus on the most important data – ORDER BY

▶ The ORDER BY clause also works for columns containing text, where the results show up in alphabetical order.

 ▶ You can reverse the order using the DESC argument (short for 'descending'). The next query sorts the table by the Animal column, where the values that are last in alphabetic order are returned first

```
query = """
    SELECT ID, Name, Animal
    FROM `bigquery-public-data.pet_records.pets`
    ORDER BY Animal DESC
"""
```

ID	Name	Animal
1	Dr. Harris Bonkers	Rabbit
2	Moon	Dog
3	Ripley	Cat
4	Tom	Cat

Order results to focus on the most important data – ORDER BY

▸ Often you'll want to look at part of a date, like the year or the day. You can do this with EXTRACT

▸ The query below returns two columns, where column Day contains the day corresponding to each entry the Date column from the pets_with_date table:

ID	Name	Animal	Date
1	Dr. Harris Bonkers	Rabbit	2019-04-18
4	Tom	Cat	2019-05-16
2	Moon	Dog	2019-01-07
3	Ripley	Cat	2019-02-23

```
query = """
    SELECT Name, EXTRACT(DAY from Date) AS Day
    FROM `bigquery-public-data.pet_records.pets_with_date`
    """
```

Name	Day
Dr. Harris Bonkers	18
Tom	16
Moon	7
Ripley	23

▸ SQL is very smart about dates, and we can ask for information beyond just extracting part of the cell

```
query = """
    SELECT Name, EXTRACT(WEEK from Date) AS Week
    FROM `bigquery-public-data.pet_records.pets_with_date`
    """
```

Name	Week
Dr. Harris Bonkers	15
Tom	19
Moon	1
Ripley	7

Organize your query for better readability – AS, WITH

▸ With all that you've learned, your SQL queries are getting pretty long, which can make them hard understand (and debug)

  ▸ You are about to learn how to use AS and WITH to tidy up your queries and make them easier to read

  ▸ To use AS in SQL, insert it right after the column you select

ID	Name	Animal	Years_old
1	Dr. Harris Bonkers	Rabbit	4.5
2	Moon	Dog	9.0
3	Ripley	Cat	1.5
4	Tom	Cat	7.8

```
query = """
    SELECT Animal, COUNT(ID)
    FROM `bigquery-public-data.pet_records.pets`
    GROUP BY Animal
"""
```

Animal	f0_
Rabbit	1
Dog	1
Cat	2

```
query = """
    SELECT Animal, COUNT(ID) AS Number
    FROM `bigquery-public-data.pet_records.pets`
    GROUP BY Animal
"""
```

Animal	Number
Rabbit	1
Dog	1
Cat	2

Organize your query for better readability – AS, WITH

▶ On its own, AS is a convenient way to clean up the data returned by your query. It's even more powerful when combined with WITH in what's called a "common table expression" (or CTE)

  ▶ CTE is a temporary table that you return within your query. CTEs are helpful for splitting your queries into readable chunks, and you can write queries against them

```
query = """
    WITH Seniors AS
    (
        SELECT ID, Name
        FROM `bigquery-public-data.pet_records.pets`
        WHERE Years_old > 5
    )
        This query is incomplete. More coming soon!
"""
```

ID	Name
2	Moon
4	Tom

This is a **CTE** named `Seniors`.
(*It is not returned by the query*.)

Organize your query for better readability – AS, WITH

▸ While this incomplete query above won't return anything, it creates a CTE that we can then refer to (as Seniors) while writing the rest of the query

  ▸ You could do this without a CTE, but if this were the first part of a very long query, removing the CTE would make it much harder to follow

  ▸ Also, it's important to note that CTEs only exist inside the query where you create them, and you can't reference them in later queries. So, any query that uses a CTE is always broken into two parts: (1) first, we create the CTE, and then (2) we write a query that uses the CTE

ID	Name	Animal	Years_old
1	Dr. Harris Bonkers	Rabbit	4.5
2	Moon	Dog	9.0
3	Ripley	Cat	1.5
4	Tom	Cat	7.8

```
query = """
    WITH Seniors AS
    (
        SELECT ID, Name
        FROM `bigquery-public-data.pet_records.pets`
        WHERE Years_old > 5
    )
    SELECT ID
    FROM Seniors
"""
```

ID
2
4

Combine data sources – JOIN, UNION

- You have the tools to obtain data from a single table in whatever format you want it. But what if the data you want is spread across multiple tables?
  - That's where JOIN comes in! JOIN is incredibly important in practical SQL workflows. So let's get started

owners table

ID	Name	Age	Pet_ID
1	Aubrey Little	20	1
2	Chett Crawfish	45	3
3	Jules Spinner	10	4
4	Magnus Burnsides	9	2
5	Veronica Dunn	8	NULL

pets table

ID	Name	Age	Animal
1	Dr. Harris Bonkers	1	Rabbit
2	Moon	9	Dog
3	Ripley	7	Cat
4	Tom	2	Cat
5	Maisie	10	Dog

Dr. Harris Bonkers is owned by **Aubrey Little.**

Moon is owned by **Magnus Burnsides.**

Ripley is owned by **Chett Crawfish.**

Tom is owned by **Jules Spinner.**

Veronica Dunn does not have a pet.
Maisie does not have an owner.

Combine data sources – JOIN, UNION

▸ Using JOIN, we can write a query to create a table with just two columns: the name of the pet and the name of the owner

  ▸ We combine information from both tables by matching rows where the ID column in the pets table matches the Pet_ID column in the owners table.

  ▸ In the query, **ON determines which column in each table to use to combine the tables**. Notice that since the ID column exists in both tables, we have to clarify which one to use. We use p.ID to refer to the ID column from the pets table, and o.Pet_ID refers to the Pet_ID column from the owners table

```
query = """
    SELECT p.Name AS Pet_Name, o.Name AS Owner_Name
    FROM `bigquery-public-data.pet_records.pets` AS p
    INNER JOIN `bigquery-public-data.pet_records.owners` AS o
        ON p.ID = o.Pet_ID
"""
```

Pet_Name	Owner_Name
Dr. Harris Bonkers	Aubrey Little
Moon	Magnus Burnsides
Ripley	Chett Crawfish
Tom	Jules Spinner

Combine data sources – JOIN, UNION

▸ To create a table containing all rows from the owners table, we use a LEFT JOIN. In this case, "left" refers to the table that appears before the JOIN in the query. ("Right" refers to the table that is after the JOIN.)


`bigquery-public-data.pet_records.owners` **AS** o **INNER JOIN** `bigquery-public-data.pet_records.pets` **AS** p

left table   *right table*

▸ Replacing INNER JOIN in the query above with LEFT JOIN returns all rows where the two tables have matching entries, along with all of the rows in the left table (whether there is a match or not)

▸ If we instead use a RIGHT JOIN, we get the matching rows, along with all rows in the right table (whether there is a match or not)

▸ Finally, a FULL JOIN returns all rows from both tables. Note that in general, any row that does not have a match in both tables will have NULL entries for the missing values. You can see this in the image below
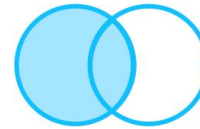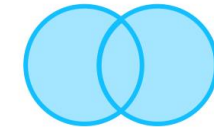
Combine data sources – JOIN, UNION

INNER JOIN
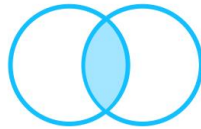
Owner_Name	Pet_Name
Aubrey Little	Dr. Harris Bonkers
Magnus Burnsides	Moon
Chett Crawfish	Ripley
Jules Spinner	Tom

LEFT JOIN

Owner_Name	Pet_Name
Aubrey Little	Dr. Harris Bonkers
Magnus Burnsides	Moon
Chett Crawfish	Ripley
Jules Spinner	Tom
Veronica Dunn	**NULL**

FULL JOIN

Owner_Name	Pet_Name
Aubrey Little	Dr. Harris Bonkers
Magnus Burnsides	Moon
Chett Crawfish	Ripley
Jules Spinner	Tom
Veronica Dunn	**NULL**
NULL	Maisie

Combine data sources – JOIN, UNION

▸ JOINs horizontally combine results from different tables. If you instead would like to vertically concatenate columns, you can do so with a UNION. The example query below combines the Age columns from both tables

  ▸ Note that with a UNION, the data types of both columns must be the same, but the column names can be different

  ▸ So, for instance, we cannot take the UNION of the Age column from the owners table and the Pet_Name column from the pets table

```
query = """"
        SELECT Age FROM `bigquery-public-data.pet_records.pets`
        UNION ALL
        SELECT Age FROM `bigquery-public-data.pet_records.owners`
        """"
```

| Age |
| --- |
| 20 |
| 45 |
| 10 |
| 9 |
| 8 |
| 1 |
| 9 |
| 7 |
| 2 |
| 10 |

Combine data sources – JOIN, UNION

▶ Unlike aggregate functions, analytic functions return a (potentially different) value for each row in the original table

- ▶ Analytic functions allow us to perform complex calculations with relatively straightforward syntax. For instance, we can quickly calculate moving averages and running totals, among other quantities

- ▶ We'd like to calculate a moving average of the training times for each runner, where we always take the average of the current and previous training sessions. We can do this with the following query:

id	date	time
1	2019-07-05	22
1	2019-04-15	26
2	2019-02-06	28
1	2019-01-02	30
2	2019-08-30	20
2	2019-03-09	22

```
query = """
    SELECT *,
        AVG(time) OVER(
                PARTITION BY id
                ORDER BY date
                ROWS BETWEEN 1 PRECEDING AND CURRENT ROW
                ) as avg_time
    FROM `bigquery-public-data.runners.train_time`
"""
```

Analytical functions

▶ All analytic functions have an OVER clause, which defines the sets of rows used in each calculation. The OVER clause has three parts:

- ▶ The PARTITION BY clause divides the rows of the table into different groups
- ▶ The ORDER BY clause defines an ordering within each partition
- ▶ The final clause known as a *window frame* clause. It identifies the set of rows used in each calculation. We can refer to this group of rows as a window
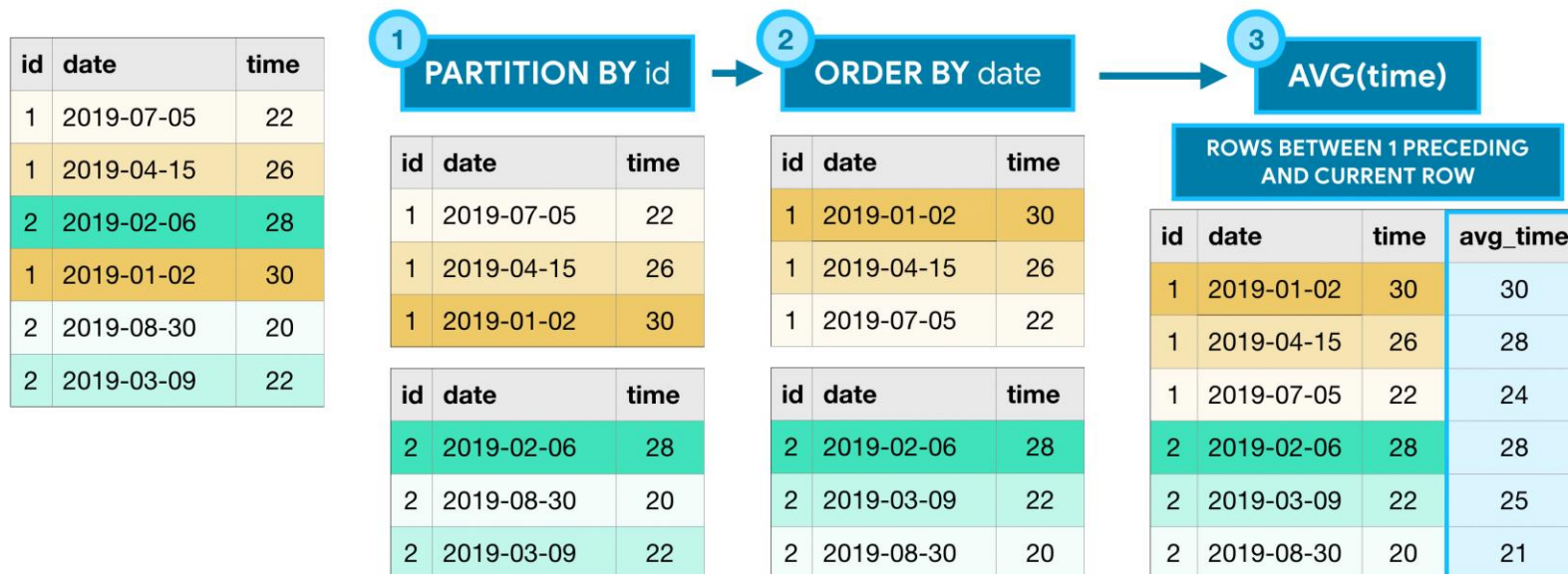
3. Data wrangling with Pandas

▶ When you have a lot of data, and it is contained in many different linked tables, it definitely makes sense to use SQL for working with it

  ▶ However, there are many cases when we have a table of data, and we need to gain some **understanding** or **insights** about this data, such as the distribution, correlation between values, etc. In data science, there are a lot of cases when we need to perform some transformations of the original data, followed by visualization

▶ Pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data

  ▶ Most of the operations in SQL can be implemented with Pandas

  ▶ Flexible group by functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data. Intuitive merging and joining data sets

 ▶ Intelligent label-based slicing, fancy indexing, and subsetting of large data sets

Data wrangling with Pandas

‣ In addition

    ‣ Easy handling of missing data in floating point as well as non-floating point data

    ‣ Flexible reshaping and pivoting of data sets

    ‣ Hierarchical labeling of axes (possible to have multiple labels per tick)

    ‣ Robust IO tools for loading data from flat files (CSV and delimited), Excel files, databases

    ‣ Time series-specific functionality: date range generation and frequency conversion, moving window statistics, date shifting, and lagging

‣ To see the important data wrangling techniques, refer to
https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf

Series

▸ Series is a sequence of values, similar to a list or Numpy array. The main difference is that series also has an index, and when we operate on series (eg., add them), the index is taken into account. Index can be as simple as integer row number (it is the index used by default when creating a series from list or array), or it can have a complex structure, such as date interval

  ▸ A DataFrame is essentially a collection of series with the same index. We can combine several series together into a DataFrame

Series
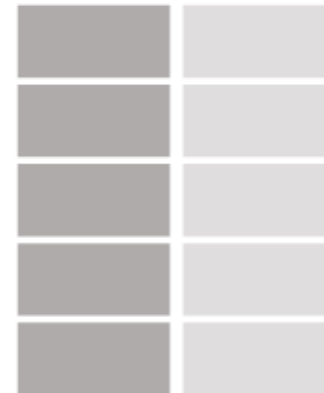
DataFrame

▸ It is a 2-dimensional data structure that can store data of different types (including characters, integers, floating-point values, categorical data and more) in columns

  ▸ It is similar to a spreadsheet, a SQL table or the data.frame in R.

    ▸ https://pandas.pydata.org/docs/getting_started/index.html

  ▸ Rows indicating records (cases) and columns indicating features (variables);
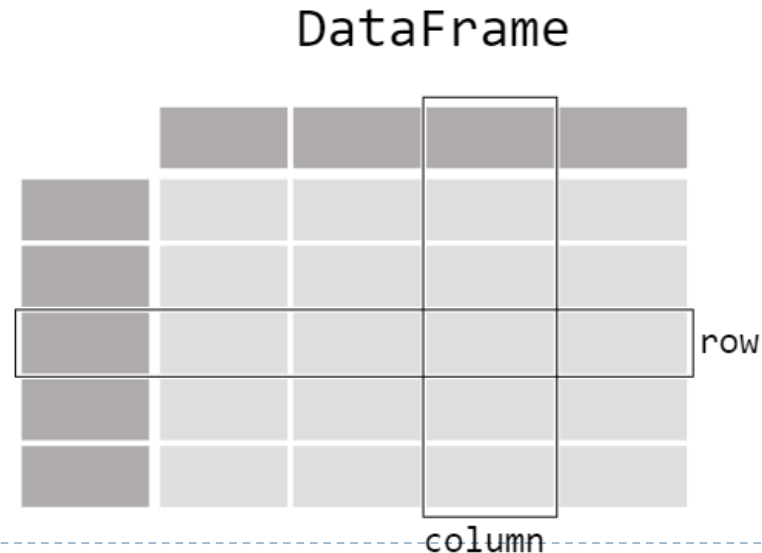
Conclusion

▸ SQL is a pretty essential part of the data scientist's toolkit.

▸ When you have a lot of data, and it is contained in many different linked tables, it definitely makes sense to use SQL for working with it

▸ However, there are many cases when we have a table of data, and we need to gain some understanding or insights about this data, such as the distribution, correlation between values, etc. In data science, there are a lot of cases when we need to perform some transformations of the original data, followed by visualization. In these cases, spreadsheet is helpful

▸ We now have basic tools for manipulating with data, we can go to understand and prepare our data in next lecture!

References

[1] Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition Chapter 1

[2] https://github.com/microsoft/Data-Science-For-Beginners/blob/main/2-Working-With-Data/05-relational-databases/README.md

[3] https://www.kaggle.com/learn/intro-to-sql

[4] https://www.kaggle.com/learn/advanced-sql

[5] Python for Data Analysis, 2nd Edition

[6] Data Science from scratch, 2nd Edition

Appendix

Resources and libraries

▸ SQL and pandas

  ▸ https://www.kaggle.com/learn/pandas

  ▸ https://www.kaggle.com/learn/intro-to-sql

  ▸ https://github.com/pingcap/awesome-database-learning

  ▸ https://15445.courses.cs.cmu.edu/fall2019/schedule.html#aug-26-2019

▸ Introduction to SQLite

  ▸ https://ucsbcarpentry.github.io/2020-01-31-UCSB-SQL/

Resources and libraries

‣ High performance dataframe for large data

  ‣ https://www.datarevenue.com/en-blog/pandas-vs-dask-vs-vaex-vs-modin-vs-rapids-vs-ray

  ‣ https://pandas.pydata.org/pandas-docs/stable/user_guide/scale.html

  ‣ https://stackoverflow.com/questions/26124417/how-to-convert-a-csv-file-to-parquet

‣ Non-relational data format

  ‣ https://github.com/microsoft/Data-Science-For-Beginners/blob/main/2-Working-With-Data/06-non-relational/README.md

  ‣ https://en.wikipedia.org/wiki/Data_lake

Sources of Data

- A data source is the initial location of where the data was generated, or where it "lives" and will vary based on how and when it was collected

  - Data generated by its user(s) are known as primary data while secondary data comes from a source that has collected data for general use. For example, a group of scientists collecting observations in a rainforest would be considered primary and if they decide to share it with other scientists it would be considered secondary to those that use it

  - Databases are a common source and rely on a database management system to host and maintain the data where users use commands called *queries* to explore the data. Files as data sources can be audio, image, and video files as well as spreadsheets like Excel. Internet sources are a common location for hosting data, where databases as well as files can be found

The shortcomings of a single table approach

▸ Begin our exploration by starting a table to store information about cities. We might start with their name and country

  ▸ Let's start to add some additional data to our burgeoning database - annual rainfall (in millimeters). We'll focus on the years 2018, 2019 and 2020. If we were to add it for Tokyo, it might look something like this

  ▸ It could take up quite a bit of storage, and is largely unnecessary to have multiple copies

City	Country
Tokyo	Japan
Atlanta	United States
Auckland	New Zealand
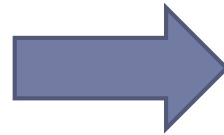
City	Country	Year	Amount
Tokyo	Japan	2020	1690
Tokyo	Japan	2019	1874
Tokyo	Japan	2018	1445

The shortcomings of a single table approach

▸ Let's try something else. Let's add new columns for each year

  ▸ While this avoids the row duplication, it adds a couple of other challenges. We would need to modify the structure of our table each time there's a new year. Additionally, as our data grows having our years as columns will make it trickier to retrieve and calculate values

  ▸ This is why we need multiple tables and relationships. By breaking apart our data we can avoid duplication and have more flexibility in how we work with our data

City	Country	2018	2019	2020
Tokyo	Japan	1445	1874	1690
Atlanta	United States	1779	1111	1683
Auckland	New Zealand	1386	942	1176

The concepts of relationships

▸ Let's determine how we want to split things up. We know we want to store the name and country for our cities, so this will probably work best in one table

  ▸ But before we create the next table, we need to figure out how to reference each city. We need some form of an identifier, ID or (in technical database terms) a *primary key*

  ▸ A primary key is a value used to identify one specific row in a table. We don't want the id to ever change as it would break the relationship

  ▸ With our cities table created, let's store the rainfall. **Rather than duplicating the full information about the city**, we can use the id. We should also ensure the newly created table has an id column as well, as all tables should have an id or primary key

city_id	City	Country
1	Tokyo	Japan
2	Atlanta	United States
3	Auckland	New Zealand

The concepts of relationships

▶ Notice the *city_id* column inside the newly created rainfall table. This column contains values which reference the IDs in the cities table. In technical relational data terms, this is called a *foreign key*; it's a primary key from another table. You can just think of it as a reference or a pointer

rainfall_id	city_id	Year	Amount
1	1	2018	1445
2	1	2019	1874
3	1	2020	1690
4	2	2018	1779
5	2	2019	1111
6	2	2020	1683
7	3	2018	1386
8	3	2019	942
9	3	2020	1176

Nested data

▸ Consider a hypothetical dataset containing information about pets and their toys. We could organize this information in two different tables (a pets table and a toys table). The toys table could contain a "Pet_ID" column that could be used to match each toy to the pet that owns it.

▸ Another option is to organize all of the information in a single table, similar to the pets_and_toys table below.

`pets` table

ID	Name	Age	Animal
1	Moon	9	Dog
2	Ripley	7	Cat
3	Napoleon	1	Fish

`toys` table

ID	Name	Type	Pet_ID
1	McFly	Frisbee	1
2	Fluffy	Feather	2
3	Eddy	Castle	3

vs.

`pets_and_toys` table

ID	Name	Age	Animal	Toy
1	Moon	9	Dog	{**Name**: McFly, **Type**: Frisbee}
2	Ripley	7	Cat	{**Name**: Fluffy, **Type**: Feather}
3	Napoleon	1	Fish	{**Name**: Eddy, **Type**: Castle}

Nested data

▸ To query a column with nested data, we need to identify each field in the context of the column that contains it:

  ▸ Toy.Name refers to the "Name" field in the "Toy" column, and

  ▸ Toy.Type refers to the "Type" field in the "Toy" column.

Nested data

▸ Now consider the (more realistic!) case where each pet can have multiple toys. In this case, to collapse this information into a single table, we need to leverage a different datatype

  ▸ We say that the "Toys" column contains repeated data, because it permits more than one value for each row. This is reflected in the table schema below, where the mode of the "Toys" column appears as 'REPEATED'.

pets table

ID	Name	Age	Animal
1	Moon	9	Dog
2	Ripley	7	Cat
3	Napoleon	1	Fish

toys_type table

ID	Type	Pet_ID
1	Frisbee	1
2	Bone	1
3	Rope	1
4	Feather	2
5	Ball	2
6	Castle	3

vs.

pets_and_toys_type table

ID	Name	Age	Animal	Toys
1	Moon	9	Dog	[Frisbee, Bone, Rope]
2	Ripley	7	Cat	[Feather, Ball]
3	Napoleon	1	Fish	[Castle]

Nested data

▸ When querying repeated data, we need to put the name of the column containing the repeated data inside an UNNEST() function.

```
query = """"
        SELECT Name AS Pet_Name,
               Toy_Type
        FROM `bigquery-public-data.pet_records.pets_and_toys_type`,
             UNNEST(Toys) AS Toy_Type
        """"
```

Pet_Name	Toy_Type
Moon	Frisbee
Moon	Bone
Moon	Rope
Ripley	Feather
Ripley	Ball
Napoleon	Castle

▸ This essentially flattens the repeated data (which is then appended to the right side of the table) so that we have one element on each row. For an illustration of this, check out the image below

`bigquery-public-data.pet_records.pets_and_toys_type`,
UNNEST(Toys) AS Toy_Type

This is not a complete query. We just illustrate what this part of the original query does.

ID	Name	Age	Animal	Toys	Toy_Type
1	Moon	9	Dog	[Frisbee, Bone, Rope]	Frisbee
1	Moon	9	Dog	[Frisbee, Bone, Rope]	Bone
1	Moon	9	Dog	[Frisbee, Bone, Rope]	Rope
2	Ripley	7	Cat	[Feather, Ball]	Feather
2	Ripley	7	Cat	[Feather, Ball]	Ball
3	Napoleon	1	Fish	[Castle]	Castle