CS-A1153

Databases for Data Science

Data Cleaning and Data Analysis in SQL

May 18, 2021

Aalto University
School of Science

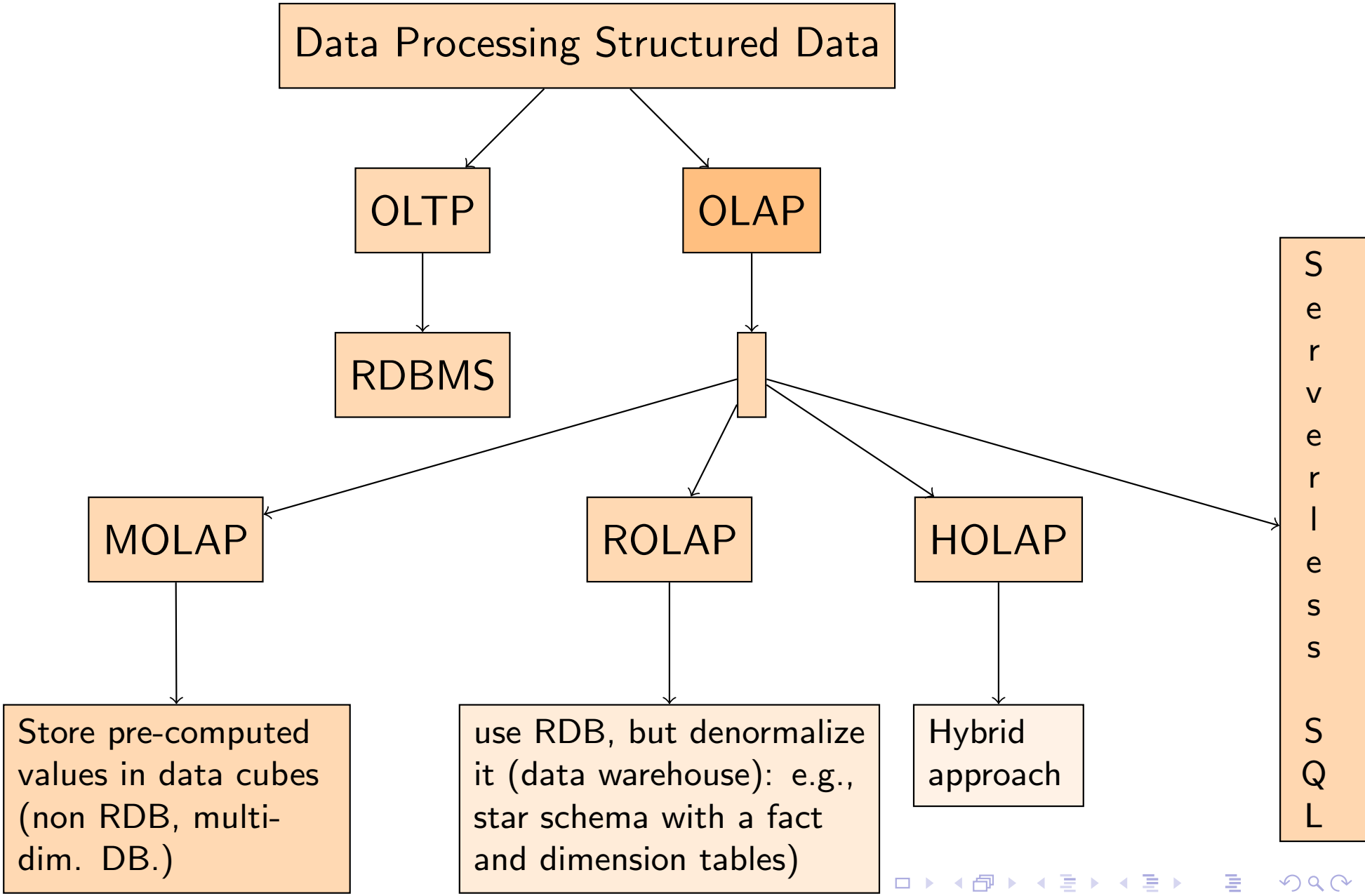
Instructor:
Dr. Sami El-Mahgary

Data Cleaning and Data Analysis

CS-A1153: Databases

Dr. Sami El-Mahgary

Aalto University
School of Science


Data Processing Structured Data
OLTP
OLAP
RDBMS
MOLAP
ROLAP
HOLAP
Serverless SQL
Store pre-computed values in data cubes (non RDB, multi-dim. DB.)
use RDB, but denormalize it (data warehouse): e.g., star schema with a fact and dimension tables)
Hybrid approach

Data and its life-cycle (1/3)

A brief extract of some types of data:

1. **1960's** *panel (longitudinal) data*: used in surveys, multiple variables/attributes on individuals' viewpoints over points **in time**.
2. **mid-1970's** *relational data*: a database schema consisting of tables based on relations. A table has a (unique) **key** and **atomic** columns, (could not combine birth date with birth place). All non-key columns in a table semantically depend on the key and only on it.
3. **late 1970's** *spreadsheet data*: tabularly organized, each entry is a new row made of the same number of columns, which can be of **different type**. Suitable for storing panel data.
4. **1990's** *spatial (GIS) data*. Typically narrow (few columns) with a huge no of rows. An example are **point clouds** (via Laser scanning).
5. **early 2000** *big data*: characterized by its **variety**: (un)structured/semistructured, its **volume** and its **velocity** (typically collected through sensors).

Types of data and its life-cycle (2/3)

Data cleaning addresses the following issues found in raw data:

1. missing values (incomplete or missing or not applicable information)
2. outliers: values beyond the expected range, that stand out.
3. correct formats (consistent data types: e.g. standard format for dates and times; numerical values for which averages are to be computed should not be of type integer).
4. other anomalies such as duplicates or near-duplicates

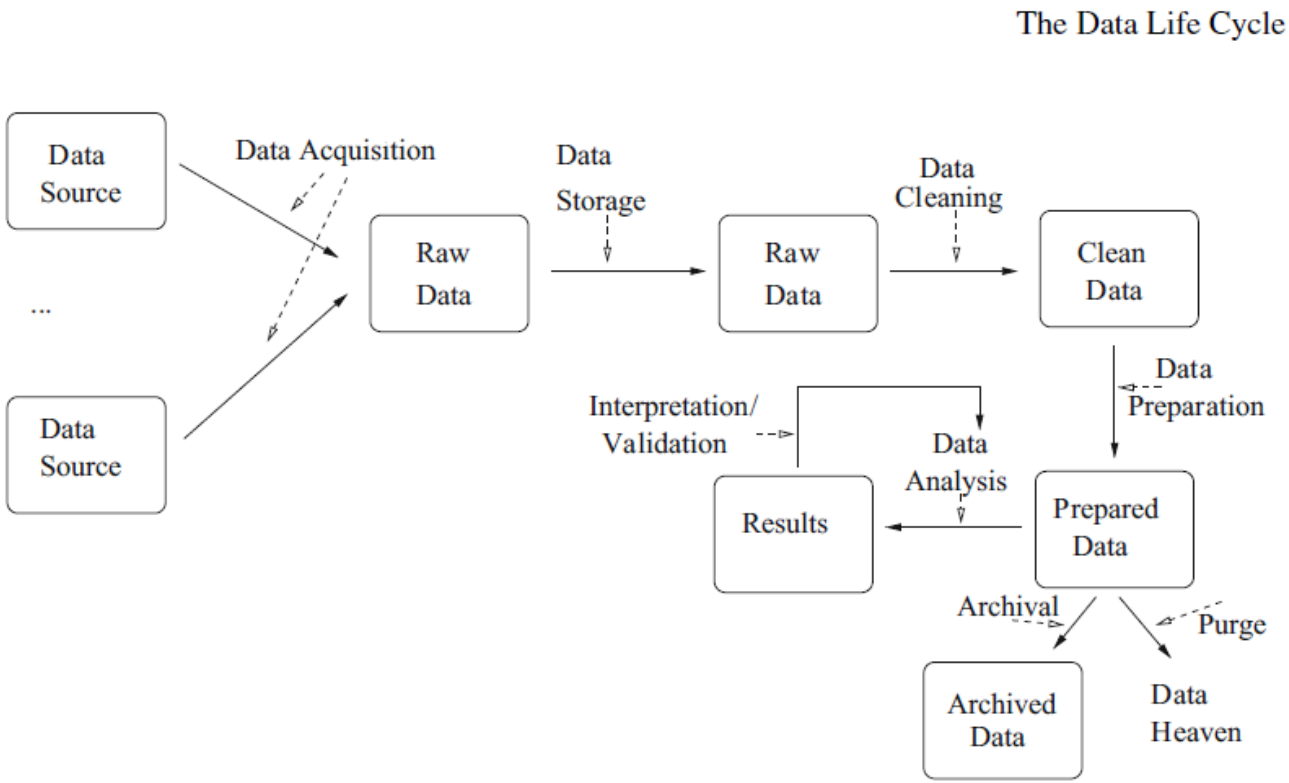
Typically, raw (un-cleaned) data is first read from a semi-structured file format, such as a CSV-file or the XML and JSON data formats. A relational database management systems (RDBMS) is for structured data and further requires:

- atomic values: each column contains only one semantic information (e.g., do not combine birtdate with birth place).
- uniquely identifying each data row through a key value

Data and its life-cycle (3/3)

Cleaned and prepared (transformed) data can be stored in a RDBMS

for manipulation and analysis via SQL and Python/pandas.



Source: Antonio Badia, *SQL for Data Science*, Springer, 2020.

Data Representations : Standard view (0/3)

An extract of data for milk batches. Composite key: MDate + Vat.
SCC = Somatic cell count (per ml for bacteria)

The dataset extract below could be an SQL-table with composite primary key `MDate,Vat`.

It is known as a wide or unstacked data representation. [1]

MDate	**Vat**	**MBatch**	**Fat%**	**Prot%**	**SCC**	**MilkLtrs**
2021-03-01	1	B1	3.11	3.50	45 000	10 500
2021-03-01	2	B2	3.07	3.56	30 000	11 000
2021-03-01	3	B2	3.07	3.56	30 000	10 000
2021-03-02	...	...	...	...	...	
2021-03-31	1	B1	3.09	3.58	47 000	11 000
2021-03-31	2	B1	3.09	3.58	47 000	11 200
2021-03-31	3	B2	3.14	3.59	35 000	10 500

[1] Badia, SQL for Data Science, Sect. 2.1.4.

Data Representations Summary (1/3)

Composite key is now: MDate + MBatch (less detail, no `Vat`).

MDate	**MBatch**	**AvgFat%**	**MaxSCC**	**SumMilkLtrs**
2021-03-01	B1	3.11	45 000	10 500
2021-03-01	B2	3.07	30 000	21 000
2021-03-02	...	...	...	...
2021-03-31	B1	3.09	47 000	22 200
2021-03-31	B2	3.14	35 000	10 500

The dataset above is a summary of the previous dataset.

It **groups** rows that belong to the same day and milk batch together.

- Note that the data works at a higher level of detail (less rows per each day).
- Numerical values have been **aggregated** (average, max., sum). Other operations could have been minimum or count.

Data Representations (2A/3)

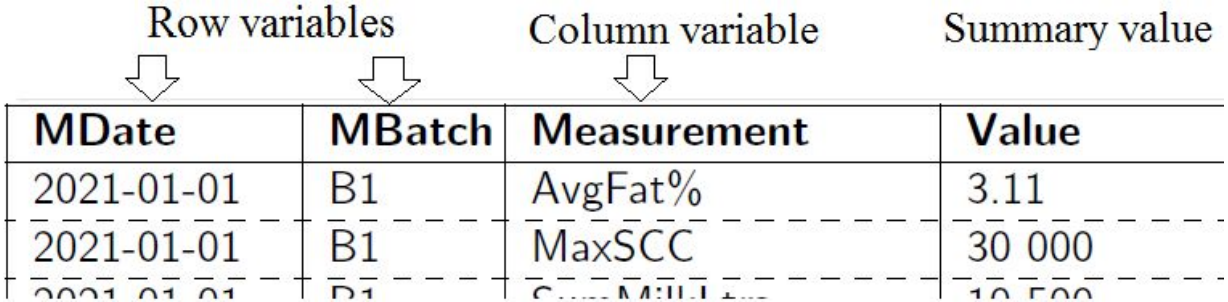
Below is a **narrow** or stacked data representation[2]

It is usually **much** less convenient for data analysis and requires more rows. *All attributes should be in the schema, not in the data*. The fact that adding a new type of measurement (e.g. Lactose%) requires just adding a new row and not restructing the table is **not** a valid reason for using this weak approach.

MDate	MBatch	Measurement	Value
2021-01-01	B1	AvgFat%	3.11
2021-01-01	B1	MaxSCC	30 000
2021-01-01	B1	SumMilkLtrs	10 500
...	.	..	...
2021-01-31	B2	MaxSCC	35 000

[2] Badia, SQL for Data Science, Sect. 2.1.4.

Data Representations (2B/3)



MDate	MBatch	Measurement	Value
2021-01-01	B1	AvgFat%	3.11
2021-01-01	B1	MaxSCC	30 000

Distinct values in column 'Measurement' are transformed into separate columns.

Stacked data (above) can often be transformed into wide data (below) through the **cross-tabulation** (cross-tab, pivot) operation.

MDate	MBatch	AvgFat%	MaxSCC	SumMilkLtrs
2021-01-01	B1	3.11	30 000	10 500
2021-01-01	B2	3.07	45 000	21 000
...	...	...	...	..
2021-01-31	B1	3.09	47 000	22 200
2021-01-31	B2	3.14	35 000	10 500

Data Representations: Still Pivoting (3/3).

MDate	MBatch	SumMilkLtrs
2021-01-01	B1	10 500
2021-01-01	B2	21 000
...	B1	10 500
2021-01-31	B1	22 200
2021-01-31	B2	10 500

The dataset above has been grouped according to columns 'MDate' and 'MBatch' and the total amount of milk used per day per batch

has been computed. Below is a cross-tab of the same data (fewer rows) using 'Mbatch' as a **column variable**.

MDate	B1	B2
2021-01-01	10 500	21 000
...	...	
2021-01-31	22 200	10 500

Some limitations of SQL: GROUPING with SQL

```sql
SELECT name FROM ChicagoEmployees,
(SELECT avg(salary) AS AvgSal
FROM ChicagoEmployees
WHERE salaried = 'T')
WHERE salary > AvgSal;
```

One of the characteristics of SQL that drives data analysts mad is that it takes a query to calculate an aggregated value. If you want to *use* it, you will have to write an *aggregated subquery*. Suppose, for instance, that you have the table *Chicago employees*, and you want to find out, among the salaried employees, which ones make more than average. One query is needed to compute the average salary, and another one to use this result and compute the answer.

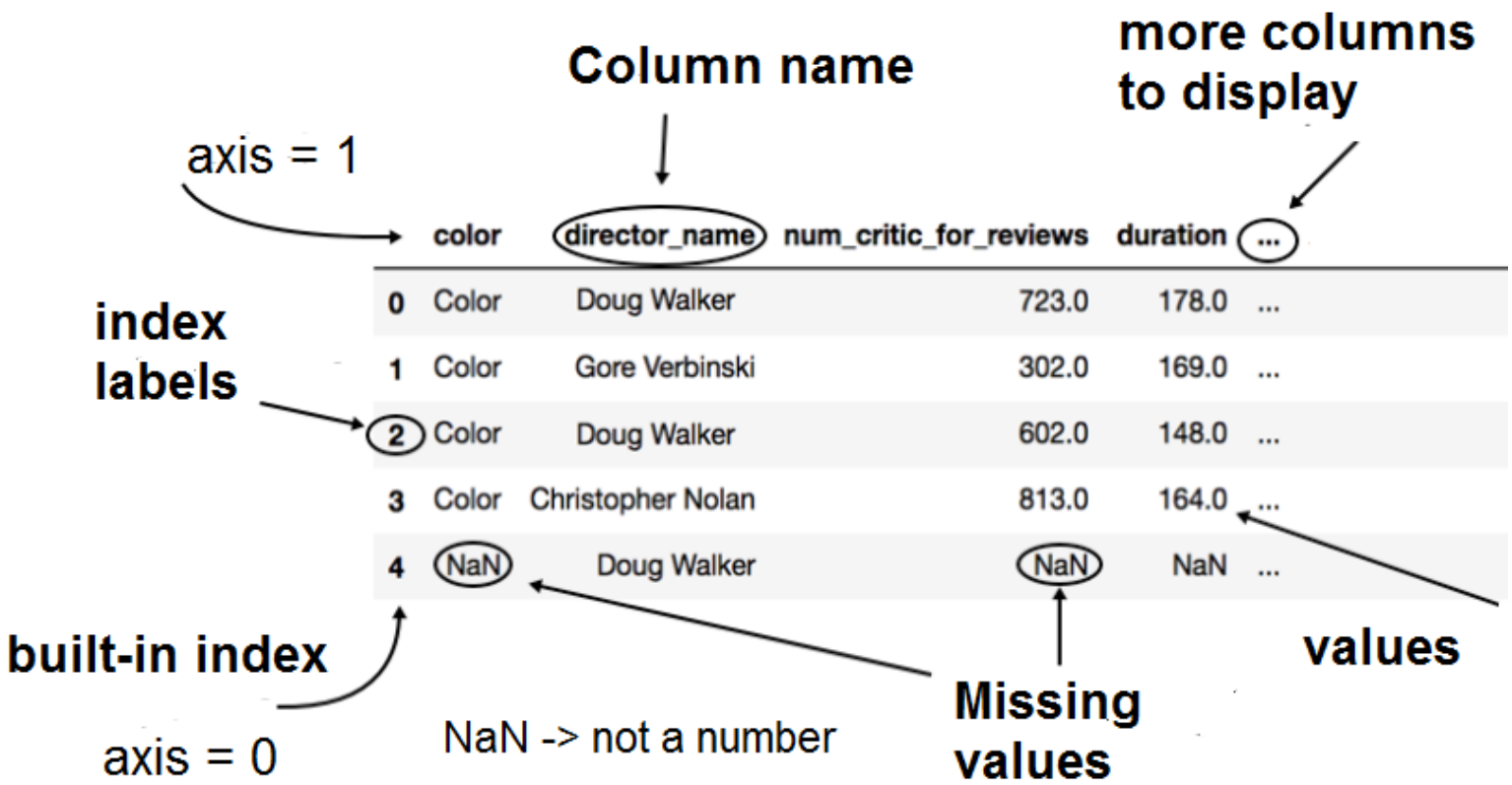
Source: Antonio Badia, *SQL for Data Science*, Springer, 2020.

Solution: Extend SQL with pandas

Some limitations of SQL: Meet the dataframe

The dataframe was originally developed for the R language

and later implemented into pandas (via numpy). In pandas, each series is an array of like-structured data that **is indexed**. A dataframe is a collection of series sharing a common index.



	color	director_name	num_critic_for_reviews	duration	...
0	Color	Doug Walker	723.0	178.0	...
1	Color	Gore Verbinski	302.0	169.0	...
2	Color	Doug Walker	602.0	148.0	...
3	Color	Christopher Nolan	813.0	164.0	...
4	NaN	Doug Walker	NaN	NaN	...

Setting up pandas and sqlalchemy in Python 1/1

```
1 import sys
2 # pandas to manipulate SQL answer set
3 import pandas as pd
4 import time,os
5 # for SQLite and other RDBMS
6 from sqlalchemy import create_engine,event,schema,Table,text
7 from sqlalchemy_utils import database_exists,create_database
8 from sqlalchemy.orm import sessionmaker
```

About pandas and sqlalchemy packages

- pandas is for manipulating data through dataframes. Sructure is very similar to an SQL-table, with built-in indexing.
- sqlalchemy is an interface between Python and RDBMSs. Instead of using drivers/code specific to a certain RDBMS, it provides a **generic** solution to interact with most popular RDBMSs.

About using pandas and SQL (1/2)

SQL shines in:

- ✓ SQL-tables are optimized for disk usage. A dataframe has no persistance as such.
- ✓ Indexing: RDBMSs were developed when memory was expensive. The B-tree indexing is very good at retrieving data from disk. A concept still useful today: memory is cheap, but datasets can be huge.
- ✓ Filtering (removing unwanted rows) and joins is what SQL is all about.
- ✓ Complex queries with SQL never require the data to fit into memory. [✗] Memory-problems with pandas are likely for very large datasets (storing even a single empty string requires about 50bytes of RAM).

About using pandas and SQL (2/2)

Pandas shines in:

- ✓ built on top of numpy (a sort of superset, not subset) supporting **contiguous memory** access (memory pointers implemented in C/Fortran). Pandas stands for **pan**el **da**ta.
- ✓ transforming grouped data into a cross-table. This is known as *pivoting* and is difficult with SQL.
- ✓ calculating basic statistics (e.g. covariances, correlations, data ranking) is simple with pandas. Data visualization is also available.
- ✓ various operations on **summarized** data are usually very fast in pandas: the data is already in memory. [✗] SQL might need to fetch most data from disk (e.g., if stored as a materialized view).

→ Combining SQL with pandas brings out the best in both.

Reading in a CSV-file 1/3

```
13 try:
14     # for Windows use \\ as separator
15     TXT_ = '.txt'; CSV_ = '.csv'; QTS_ = '"'; SLASH_ = '\\'
16     #-------------------------------------
17     # below is an extract from  Harvard Flights data
18     # https://doi.org/10.7910/DVN/HG7NV7
19     #-------------------------------------
20     FLIGHTS_      =  'flight_short'
21     AIRLINES_     = 'carriers';AIRPORTS_     = 'airports'
22     SUPPRESS_QTS = True; SUM_FDATA    = 'SummaryFlight'
23     SQLITE_SRV    = 'sqlite:///'; DB_NAME_ = 'Flights.db3'
```

Reading in a CSV-file 2/3

```
24      #-------------------------------------
25      # R E A D I N G   3  C S V   F I L E S
26      #-------------------------------------
27      df_             = pd.read_csv(
28      my_path + FLIGHTS_,sep=';',comment='#',dtype='unicode')
29      df_carriers   = pd.read_csv(
30      my_path + AIRLINES_,sep=';',comment='#',dtype='unicode')
31      df_airports   = pd.read_csv(
32      my_path + AIRPORTS_,sep=';',comment='#',dtype='unicode')
33      # ** Analyzing  the Flight Dataset **
34      print(list(df_.columns))
35      print(list(df_.dtypes))
36      print("Analyzed, found " +  str(
37      df_.shape[0]) + " rows; " + str(df_.shape[1]) + " cols.")
```

Reading in a CSV-file 3/3

Info on the data that was just read: column names and types (note the type 'object' ('O') in place of string) and no of rows and columns.

```
reading //home//cgrp02//flight_short.csv
['Year', 'Month', 'DayofMonth', 'DayOfWeek', 'DepTime',
'ArrTime', 'UniqueCarrier', 'FlightNum', 'TailNum',
'AirTime', 'DepDelay', 'ArrDelay', 'Origin', 'Dest',
'Distance', 'Cancelled', 'CancellationCode', 'Diverted',
'CarrierDelay', 'WeatherDelay', 'NASDelay', 'SecurityDelay',
'LateAircraftDelay']
[dtype('O'), dtype('O'), dtype('O'), dtype('O'), dtype('O'),
dtype('O'), dtype('O'), dtype('O'), dtype('O'), dtype('O'),
dtype('O'), dtype('O'),dtype('O'), dtype('O'), dtype('O'),
dtype('O'), dtype('O'), dtype('O'), dtype('O'), dtype('O'),
dtype('O'), dtype('O'), dtype('O')]
Analyzed, found 14698 rows; 23 cols.
```

Cleaning the RAW CSV-file (1/2)

Cleaning is making sure the data format is correct throughout,

identifying incomplete or erroneous data and taking appropriate action. Also includes renaming columns and removing unnecessary ones.

- Check that numeric columns are indeed numeric (integer or decimal).
- Outliers: identify any values which clearly differ from other values in the same column. Decide on whether **whole row** needs to be removed.
- Missing values: (avoid if possible, nulls are dull) can mean one of several things. For a numeric column on which the average (or min/max) is computed, be sure to check whether or not null values should be assigned a value of zero.

Cleaning the RAW CSV-file (2/2)

```
38 #  rename 2 columns. inplace = True means no copy is made.
39     df_.rename(columns={
40     'Year':'FlightYear','Month':'FlightMonth'},inplace = True)
41
42 # Have made sure that following nulls can be considered as 0
43 # fill the NaN values with zeroes and then
44 # convert the column to the required type
45     df_['DepDelay'] = df_['DepDelay'].fillna(0).astype(int)
46     ...
47 # to calculate avg distance, make sure it is decimal
48     df_['Distance'] = df_['Distance'].fillna(0).astype(float)
```

C

About indexes 1/1

```
49 # A dataframe does have an explicit internal index, but we nee
50 #  a separate column to act as an index in the SQL-table.
51 # We create one in the 1st posit. using a
52 # sequence of numbers (size of df_) and call it 'PK'.
53 df_.insert (loc = 0,column = 'PK',value = range(len(df_)))
```

About indexes

Pandas indexing does not transfer to SQL tables. So when working with both dataframes and SQL tables, it's helpful to have a separate column as an index.

Connecting with Python to a serverless DB: SQLite

```
54 # Connecting to SQLite database thru SQLAlchemy:
55     engine        = create_engine(
56     SQLITE_SRV + my_path + DB_NAME_, echo=False)
57     sqlite_conn   = engine.connect()
58     if not sqlite_conn:
59         print("DB connection is not OK!")
60         exit()
61     else:
62         print("DB connection is OK.")
```

Using SQLAlchemy makes it easier to:

- reuse the same code when changing databases.
- connect to SQLite3. Most other RDBMS can be connected to through an additional driver.
- We're using SQLAlchemy through the so-called core or SQL option.

Persistence: From a dataframe to an SQL (SQLite) table

```
65 # a dataframe  stores its data in memory -->
66 # generate SQL tables using the to_sql() command.
67     flights_sqltbl  = "Flights"; airports_sqltbl = "Airports"
68     df_.to_sql(
69     flights_sqltbl, sqlite_conn, if_exists='replace')
70     df_airports.to_sql(
71     airports_sqltbl, sqlite_conn, if_exists='replace')
```

Note the use of *sqlite_conn*, the handle to the DB.

A dataframe can be made persistent and converted to an SQL-table with **one line of code**.

Querying from an SQL table to a dataframe (1/2)

```
72 # force pandas to display enough columns
73 pd.set_option('display.max_columns', 11)
74 # define query, use sub-query to display airport name
75 # and avoid use of joins
76     sql_ = """
77           SELECT PK,FlightMonth,UniqueCarrier AS Carrier,
78             (SELECT airport FROM Airports
79               WHERE iata = Origin) As NameOrigin,
80             (SELECT airport FROM Airports
81               WHERE iata = Dest) AS NameDest FROM Flights
82         """
83 # read_sql_query: get results of SQL-query in a DataFrame.
84     tx_ = pd.read_sql_query(sql_,sqlite_conn)
85     print(tx_)
```

Querying from an SQL table to a dataframe (2/2)

Results of the previous query

	PK	FlightMonth	Carrier	NameOrigin	NameDest
0	0	1	WN	William P Hobby	Adams
1	1	1	WN	William P Hobby	Midland International
2	2	1	WN	William P Hobby	Midland International
3	3	1	WN	William P Hobby	Orlando International
4	4	1	WN	William P Hobby	Orlando International

- Note the leftmost column: the index-column is automatically generated by pandas
- The column `PK` was added to provide a single column primary key for the SQL-table.
- Note the airports which are meaningful plain text name.

Getting the big picture: GROUPING with Pandas (1/6)

```
86 # Group based on previous qry.
87 grp_1 = tx.groupby(['FlightMonth','Carrier']).agg(
88 {'PK':"count"})
89 print(grp_1)
```

The aggregate functions are as in SQL, but average is known as *mean*.

Unlike with SQL, pandas by default excludes rows where a grouping column is Null (can be avoided using *dropna=False*).

		PK
FlightMonth	Carrier	
1	CO	8
	FL	19
	HA	14
	WN	1174
	XE	249
2	CO	10
	FL	8
	HA	36

Getting the big picture: GROUPING with Pandas (2/6)

The previous pandas query is equivalent to the SQL-query below

The SQL query processor may need to first retrieve all/most rows first from disk, then perofrm the summary.

```
1 SELECT FlightMonth, Carrier, COUNT (PK) FROM Flights
2 GROUP BY FlightMonth, Carrier
```

FlightMonth	Carrier	PK
1	CO	8
	FL	19
	HA	14
	WN	1174
	XE	249
2	CO	10
	FL	8
	HA	36
	WN	13180

Getting the big picture: GROUPING with Pandas (3/6)

```
92 #rank the results in descending order
93   print(grp_1.rank(ascending = False))
```

Using rank() we can transform the aggregate data so that

instead of the actual no of flights, we have the rank of each airline Useful when numerically assessing the quality/efficiency.

		PK
FlightMonth	Carrier	
1	CO	8.5
	FL	5.0
	HA	6.0
	WN	2.0
	XE	3.0
2	CO	7.0
	FL	8.5
	HA	4.0

Getting the big picture: GROUPING with Pandas (4/6)

```
94  # generate a boolean series mask for months 1 and 2
95      mask1 = grp_1['FlightMonth'] == '1'
96      mask2 = grp_1['FlightMonth'] == '2'
97  # now get 2 dataframes, one for each month
98      grp_1A = grp_1[mask1];grp_1B = grp_1[mask2]
99  # one extra, row drop it. Still extra cols.
100     grp_1A = grp_1.head(-1)
101     print(grp_1A)
102 # rename the columns
103     grp_1A = grp_1A.rename({'PK': 'PK1'}, axis=1)
104     grp_1B = grp_1B.rename({'PK': 'PK2'}, axis=1)
```

	FlightMonth	Carrier	PK
0	1	CO	8
1	1	FL	19
2	1	HA	14
3	1	WN	1174

Getting the big picture: GROUPING with Pandas (5/6)

```
105     print(grp_1A);print(grp_1B)
106 # extract last column from each df (columns start at 0)
107     col_1A =  grp_1A[grp_1A.columns[2]]
108     col_1B =  grp_1B[grp_1B.columns[2]]
109     widedata = pd.concat(
110     [col_1A.reset_index(drop=True), col_1B.reset_index(
111     drop=True)], axis= 1)
112 # compute correlation coefficient ('PK1', 'PK2')
113     correlation = widedata['PK1'].corr(widedata['PK2'])
```

```
0       8
1      19
2      14
3    1174
Name: PK1, dtype: int64
5      10
6       8
7      36
8   13180
Name: PK2, dtype: int64
```

Getting the big picture: GROUPING with Pandas (6/6)

```
114 print(widedata); print(correlation)
```

The concatenated dataframe (4 rows) is followed by the correlation value (nearly 1) between columns PK1 and PK2.

```
     PK1     PK2
0      8      10
1     19       8
2     14      36
3   1174   13180
0.9999679093153944
```

Getting the big picture: Pivoting with Pandas (1/2)

```
118   # pivoting to make columns out of a
119   # categorical variable: here using airline code.
120       mypivot = pd.pivot_table(
121       tx_, values='PK',index=['FlightMonth'],
122       columns = tx_.Carrier.values, aggfunc = 'count').fillna(0)
123       print(mypivot)
124 # below using grp_1: already grouped data, gives same result
125       mypivot = pd.pivot_table(
126       grp_1, values='PK',index=['FlightMonth'],
127       columns=grp_1.Carrier.values).fillna(0)
```

FlightMonth	CO	FL	HA	WN	XE
1	8.0	19.0	14.0	1174.0	249.0
2	10.0	8.0	36.0	13180.0	0.0

Getting the big picture: Pivoting with Pandas (2/2)

To get rid of decimals, force type-conversion

to an integer by appending *.astype(int)*

```
128     ...).fillna(0).astype(int)
129     print(mypivot)
```

FlightMonth	CO	FL	HA	WN	XE
1	8	19	14	1174	249
2	10	8	36	13180	0

Summary of Data Analysis with RDBMS

- OLTP (Online-transaction processing) is row-oriented and expects speedy transactions that guarantee the ACID properties.
- OLAP (Online-analytical processing) is about accessing large amounts of read-only data quickly, usually only specific columns. Can often work with data that is no longer 100% up-to-date.
- Data-warehouses rely on de-normalizing the data (i.e., the star schema) for speed. But further normalization, e.g., 6NF, actually results in narrow tables that are very quick to load.
- An example of 6NF tables are catalog or look-up tables that consist of a primary key and a single non-key column.
- Aggregation and pivoting can be performed easily and rapidly on a normalized RDBMS via pandas. In most cases, no need to keep a separate, summarized data warehouse (or MDB).

SQL and Python

CS-A1153: Databases

Dr. Sami El-Mahgary and responsible TAs

Connect to virtual server

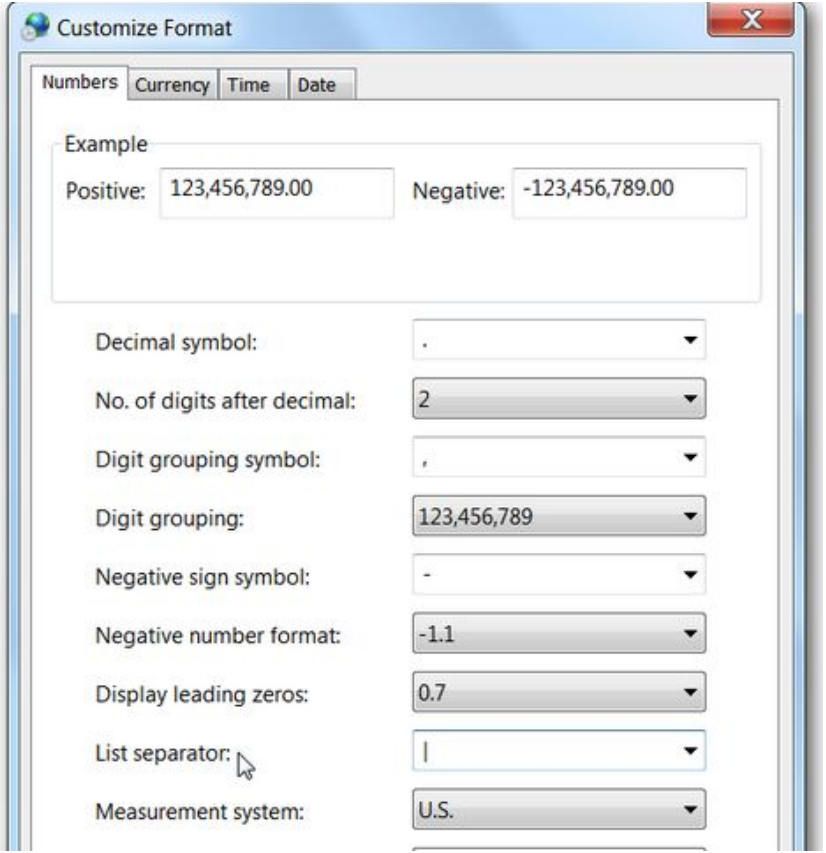
- Instruction for establish VPN connection to an Aalto network (link)
- Further instruction for connect to our course virtual server as well as preparing Thonny Python environment can be found from MyCourse → Platforms and Tools → Virtual Server → ‘this file’

Prepare sample data

- Sample data can be found from MyCourse → Group Projects
- The data is given as an Excel file made of sheets and needs to be converted into a set of CSV files: each sheet is a separate CSV file.
- A few notes on file conversion:
  - Only the active sheet can be saved. In order to save all sheets from the given excel file, you will need to save them individually as CSV using different file name for each.
  - For LibreOffice and OpenOffice, make sure that the **'Field delimiter'** is **comma** and **'String delimiter'** is **double quote**.

Prepare sample data (cont.)

- For MS Excel:
  - **For Windows 10:** Control Panel → Region → Additional Settings... → Customize Format
  - **For Windows 7???**: Edit your Regional Settings for Windows (*Regions and Language*). On the Formats tab, choose *Additional Settings* ...
  - Change the *List separator* to ***comma***



Transfer files between local computer and virtual server

Windows

1. Open a SFTP session in MobaXTerm and specify your group's folder: Enter the virtual server (*db-course.cs.aalto.fi*) into **Remote host**, and your *Aalto User ID* into **Username**. Under Advanced Sftp settings, enter */home/cgrpxx/* into **Remote startup folder**, where *xx* is your **group number** (e.g. 03). Press **OK**, then enter your password.
2. Upload/ download file(s) to/from folder: Click the green/blue arrow next to the directory

Transfer files between local computer and virtual server

MacOS

Step 1: Download FileZilla for Mac.

Step 2: Open FileZilla and enter:

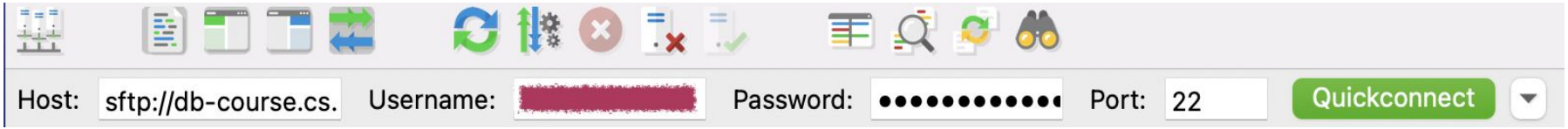
Host: sftp://db-course.cs.aalto.fi

Username: *Your username*

Password: *Your password*

Port: 22

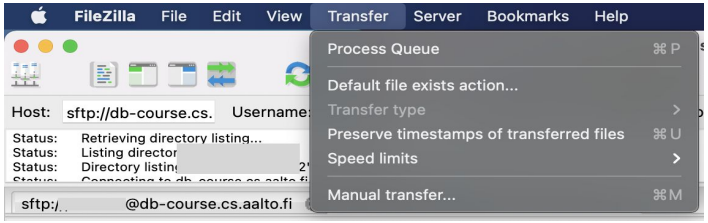
Step 3: Choose 'Quickconnect'



Transfer files between local computer and virtual server

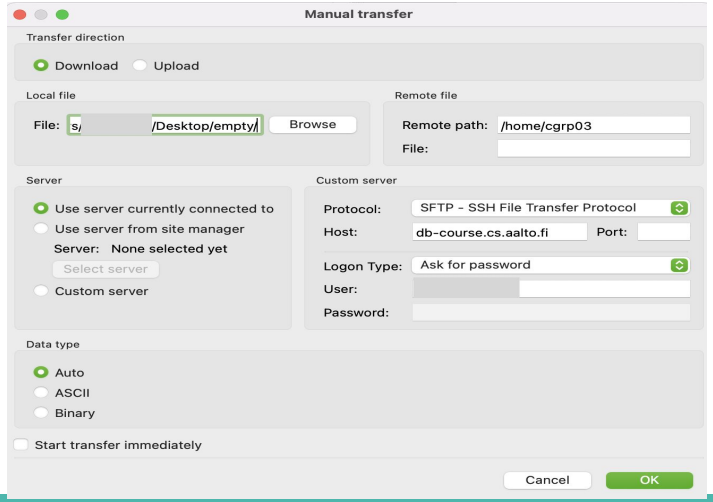
Mac OS (cont.)

Step 4: Choose ‘Transfer’ => ‘Manual transfer’



Step 5:

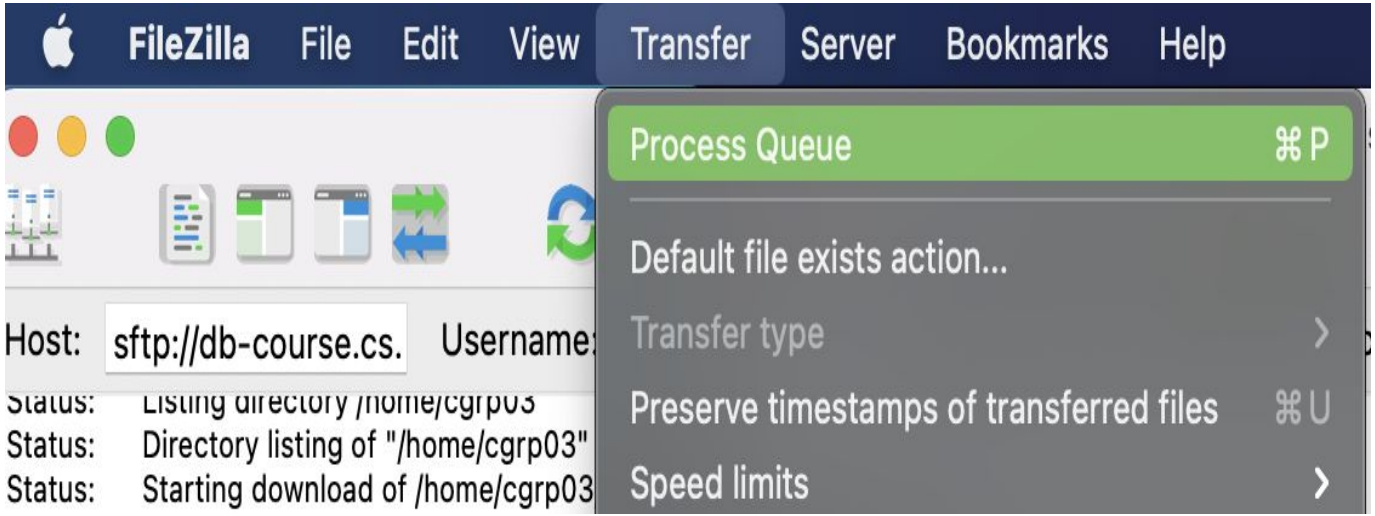
- Select ‘Download’ or ‘Upload’
- Change local and remote file paths
- Add remote filename
- Choose ‘OK’



Transfer files between local computer and virtual server

Mac OS (cont.)

Step 6: Choose ‘Transfer’ => ‘Process Queue’. Done.



Transfer files between local computer and virtual server

Linux

- Download from server:

```
scp <aalto-username>@<server-name>:/path/to/file /path/to/destination/folder
```

- Upload to server:

```
scp /path/to/file <aalto-username>@<server-name>:/path/to/destination/folder
```

Code walk-through

You are given a new example Python script using a **flights data set**. The table structures are as follow:

- airports (**iata**, airport, city, state, country, lat, long)
- carriers (code, description)
- flight_shorts (Year, Month, DayofMonth, DayOfWeek, DepTime, ArrTime, UniqueCarrier, FlightNum, ...)

Prerequisite: install matplotlib from Tools → Manage packages

After the example, you will be able to do these tasks using Python, SQLAlchemy and Pandas:

1. Read CSV files into a pandas dataframe and re-format a dataframe table (e.g. rename columns, Indexing)
2. Connect to the SQLite database and use it to populate SQL-tables via pandas dataframes
3. Perform SQL queries via pandas
4. Grouping and aggregating data
5. Basic data analysis tasks

Read CSV-files and re-format

- Read CSV files into a pandas dataframe:

```
pd.read_csv(file_path,  sep=',',comment='#',dtype='unicode')
```

- Re-format a dataframe table:
  - Rename columns:

```
df_.rename(columns={'Year':'FlightYear','Month':'FlightMonth'},inplace = True)
```

  - Deal with missing values:

```
df_['DepDelay'].fillna(0).astype(int)
```

  - Indexing:

```
df_.insert(loc = 0,column = 'PK',value = range(len(df_)))
```

Connect to SQLite and populate SQL-tables

- Connect to the SQLite database:

```
engine       = create_engine('sqlite:///' + my_path + DB_NAME_, echo=False)
sqlite_conn  = engine.connect()
```

- Using `sqlite_conn` to populate SQL-tables via pandas dataframes:

```
df_.to_sql(flights_sqltbl, sqlite_conn, if_exists = 'replace')
```

Perform SQL queries via pandas

Querry: Display PK, FlightMonth, Carrier and Name of destination airports

```
sql_ = """
       SELECT PK,FlightMonth,UniqueCarrier AS Carrier,
       (SELECT airport FROM Airports WHERE iata = Dest) AS NameDest FROM Flights
       """

tx_ = pd.read_sql_query(sql_,sqlite_conn)
```

	PK	FlightMonth	Carrier	NameDest
0	0	1	WN	Adams
1	1	1	WN	Midland International
2	2	1	WN	Midland International
3	3	1	WN	Orlando International
4	4	1	WN	Orlando International
...	...	...	...	...
14693	14693	2	WN	Fort Lauderdale-Hollywood Int'l
14694	14694	2	WN	William P Hobby
14695	14695	2	WN	William P Hobby
14696	14696	2	WN	William P Hobby
14697	14697	2	WN	William P Hobby

[14698 rows x 4 columns]

Basic data analysis tasks

- **Grouping and aggregating data** - group the result of previous query by FlightMonth and Carrier; and display the number of flights each airlines had for that month

```
grp_1 =
tx_.groupby(['FlightMonth','Carrier'],as_index=False).
agg({'PK':"count"})
```

	FlightMonth	Carrier	PK
0	1	CO	8
1	1	FL	19
2	1	HA	14
3	1	WN	1174
4	1	XE	249
5	2	CO	10
6	2	FL	8
7	2	HA	36
8	2	WN	13180

- **Ranking** - give the rank for each airlines such that the one with the most flights has rank 1.

```
grp_1['ranked_flight_num'] =
grp_1['PK'].rank(ascending = False)
```

	FlightMonth	Carrier	PK	ranked_flight_num
0	1	CO	8	8.5
1	1	FL	19	5.0
2	1	HA	14	6.0
3	1	WN	1174	2.0
4	1	XE	249	3.0
5	2	CO	10	7.0
6	2	FL	8	8.5
7	2	HA	36	4.0
8	2	WN	13180	1.0

Basic data analysis tasks (cont.)

- **Correlation** - Compute the correlation on the number of flights each airlines had between 1st and 2nd month
  - Separate `grp_1` into 2 columns (month 1 and month 2)
  - Create a new dataframe by concatenating the 2 columns (Note: we get rid of the built-index by setting `reset_index(drop=True)`)

```
widedata = pd.concat([col_1A.reset_index(drop=True),
col_1B.reset_index(drop=True)], axis= 1)
```

  - Calculate pairwise correlation of columns

```
correlation = widedata['PK1'].corr(widedata['PK2'])
```

```
    PK1    PK2
0     8     10
1    19      8
2    14     36
3  1174  13180
Correlation = 0.9999679093153944
```

Basic data analysis tasks (cont.)

FlightMonth	CO	FL	HA	WN	XE
1	8	19	14	1174	249
2	10	8	36	13180	0

- **Pivoting** - summarize the data from `grp_1` using a pivot table

```
mypivot = pd.pivot_table(grp_1,
          values='PK',index=['FlightMonth'], columns=grp_1.Carrier.values)
          .fillna(0).astype(int)
```

OR

```
mypivot = pd.pivot_table(tx_, values='PK',index=['FlightMonth'],
     columns=tx_.Carrier.values, aggfunc='count').fillna(0).astype(int)
```

- **Plotting** - using matplotlib and plot a bar-plot from pivot-table

```
myplot = mypivot.plot.bar(title ='Airlines 2008',legend=True, fontsize=11)
myplot.set_ylabel("No flights",fontsize=11)
plt.show()
```

Uncommitted Updates and Isolation Levels

CS-A1153: Databases

Dr. Sami El-Mahgary

Aalto University
School of Science

Start of Part IV

UnCommitted Updates and Isolation Levels

- Running transactions in a serial order is always safe (no anomalies).
- Running transactions in a serial order is not possible in the real world.
- Running transactions in parallel (concurrently) while minimizing anomalies and maximizing performance is a good goal.

SQL IV UnCommitted Updates and Isolation Levels

To model operations in a transaction, we use a notation known as the Key-Range Model[3]

- The beginning of transaction T_1 is indicated as B_1.
- The operation of a transaction T_1 that reads a value v from (primary) key x is denoted as $R_1[x, v]$ or for short, $R_1[x]$ (the value read is unimportant).
- The operation of a transaction T_1 that inserts a value v into (primary) key x is denoted as $I_1[x, v]$ or for short, $I_1[x]$ (the actual value inserted is unimportant).
- The operation of a transaction T_1 that deletes a row with (primary) key x is denoted as $D_1[x]$.
- A transaction T_1 that updates key x with a new value v and commits is denoted as $B_1 D_1[x] I_1[x, v] C_1$ or for short, $B_1 D_1[x] I_1[x] C_1$.

[3] See S. Sippu and E. Soisalon-Soininen, *Transaction Processing* , Springer 2015

SQL IV UnCommitted Updates (1/5).

What is a dirty value?

- A data item x is *dirty* or uncommitted at a certain moment if it was last updated (inserted, written or deleted) by a transaction T that is still active and thus it is possible that T will still change (i.e., delete, write or insert) x.
- In order to avoid any concurrency anomalies, if two transactions T_1 and T_2 are reading and writing the same data item x concurrently, then the updates of transaction T_1 should be serialized to occur either entirely before or entirely after the operations of T_2 on item x (or vice versa).
- In a serializable schedule, the transactions may be run concurrently, but without anomalies and so that the result is as if the transactions *were run* in a serial order.

SQL IV UnCommitted Updates (2/5).

Anomalies

In a concurrent schedule, the isolation of transactions can be violated in several ways. The isolation violations can be classified into three isolation anomalies: dirty writes, dirty reads and unrepeatable reads.

- A transaction T_2 does a *dirty write* if T_2 updates data items satisfying condition C when some data item satisfying C has an uncommitted update by another transaction T_1.
- A transaction T_2 does a *dirty read* if T_2 reads data items satisfying condition C when some data item satisfying C has an uncommitted update by another transaction T_1.

SQL IV UnCommitted Updates (3/5).

In the following, the condition C in the definitions is 'with key x'.

Dirty write: the change (deletion) made by T_1 cannot be undone since key x exists and is **committed**.

The action $I_2[x]$ by transaction T_2 is a dirty write in the schedule $B_1 D_1[x] B_2 I_2[x] C_2$ ($CRASH!$)

Dirty read: reading uncommitted data

The action $R_2[x]$ by transaction T_2 is a dirty write in the schedule $B_1 I_1[x] B_2 R_2[x]...$ (if T_1 aborts, $\rightarrow$ T_2 left with invalid data)

Unrepeatable read: $\rightarrow$ different results if value read twice (missing key)

The action $R_1[x]$ by transaction T_1 is an unrepeatable read in the schedule $B_1 R_1[x] B_2 D_2[x]...$

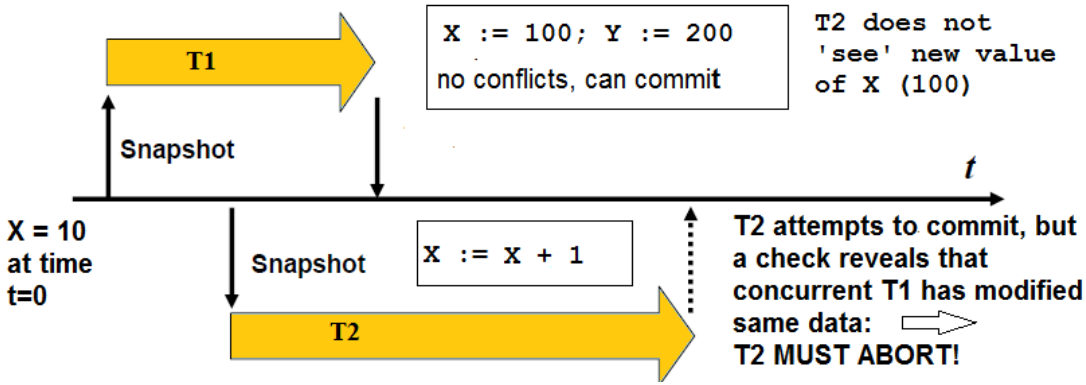
SQL IV Isolation Levels (4/5).

All levels use long-duration write locks (to avoid dirty writes).

Level Name	**ANSI?**	**Anomalies Prevented**	**Notes**
Read Uncommitted.	Yes	Dirty Write (DW)	Sees uncommitted values of other transactions: use only for long analytic. queries if approx. answer is OK. No read locks.
Read Committed[4]	Yes	DW + Dirty Read (DR)	Suits all (read-only) queries. Short-durat. read locks.
Repetable Read*	Yes	DW, DR + unrepeatable read	Phantom tuples possible. Long-durat. (entire transaction) read locks
Serializable	Yes	All	Uses also index locks.

[4] Implemented through snapshots in Oracle and in DB2 (known as Currently Committed)

SQL IV Isolation Levels: Snapshot isolation (SI) (5/5).



SI has some advantages over the Serializable isolation level.

- Always reading from a snapshot guarantees that read-only transactions will never abort.
- A read-only transaction T_i will only see values that were committed before it started.
- A (long) read-only transaction will never force a write transaction to abort.

Snapshot Isolation sounds very appealing, however its application may lead to non serializable executions. These executions result in consistency anomalies that are specific to Snapshot Isolation, such as the write-skew anomaly.

SQL IV Isolation Levels Summary.

```
1 SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

- Each RDBMS offers a default isolation level (in IBM's DB2 it is equivalent to Snapshot Isolation (Currently Committed).
- For long-running read only queries, the isolation level may be reduced to READ UNCOMMITTED as done above.
- Running a strict isolation level reduces/eliminates anomalies, but at the expense of performance.
- In some transaction intensive (OLTP) environments, thousands of transactions may be running simultaneously.
- Snapshot isolation is a preferred solution to reducing the use of locks while maintaining good performance.

End of Part IV

Most slides courtesy of S. Sippu and E. Soisalon-Soininen[5].

[5] S. Sippu and E. Soisalon-Soininen, *Transaction Processing*, Springer 2015