Data Wrangling and Data Analysis
Introduction
Relational Model, Schemas, Data Extraction in SQL and Python

Hakim Qahtan

Department of Information and Computing Sciences

Utrecht University

Utrecht University

Join Operation

- **JOIN** operations take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **FROM** clause

Topics for Today

- SQL (Continue)
- Data Extraction in Python

Join Operation (Cont.)

- We will consider the following relations in the few coming slides
- Relation *course*

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-490	Game Design	Comp. Sci.	4
CS-315	Boolean Algebra	Comp. Sci.	3

- Relation *prereq*

course_id	prereq_id
BIO-301	BIO-101
CS-490	CS-101
CS-347	CS-201

- Note that:
  - *prereq* information is missing for course CS-315
  - *course information is missing for course CS-347*

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.

Left Outer Join

```
SELECT *
FROM course
LEFT OUTER JOIN prereq
ON course.course_id = prereq.course_id
```

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-490	Game Design	Comp. Sci.	4	CS-101
CS-315	Boolean Algebra	Comp. Sci.	3	*null*

Right Outer Join

```
SELECT *
FROM course
RIGHT OUTER JOIN prereq
ON course.course_id = prereq.course_id
```

course_id	prereq_id	title	dept_name	credits
BIO-301	BIO-101	Genetics	Biology	4
CS-490	CS-101	Game Design	Comp. Sci.	4
CS-347	CS-201	*null*	*null*	*null*

- Remember:
  - The order of the attributes in a relation has no meaning

Full Outer Join

```
SELECT *
FROM course
FULL OUTER JOIN prereq
ON course.course_id = prereq.course_id
```

course_id	prereq_id	title	dept_name	credits
BIO-301	BIO-101	Genetics	Biology	4
CS-490	CS-101	Game Design	Comp. Sci.	4
CS-347	CS-201	*null*	*null*	*null*
CS-315	*null*	Boolean Algebra	Comp. Sci.	3

Inner Join

```
SELECT *
FROM course
INNER JOIN prereq
ON course.course_id = prereq.course_id
```

course_id	prereq_id	title	dept_name	credits
BIO-301	BIO-101	Genetics	Biology	4
CS-490	CS-101	Game Design	Comp. Sci.	4

- Question:
  - What is the difference between the above JOIN and the right/left outer join

String Operations

- SQL includes a string-matching operator for comparisons on character strings.
- The operator **LIKE** uses patterns that are described using two special characters
  - Percent (%). The % character matches any substring.
  - Underscore (_). The _ character matches any character.
- Example: find the names of all instructors whose name includes the substring “Van der”

```
SELECT DISTINCT name
FROM instructor WHERE name LIKE ‘%Van der%’
```

String Operations (Cont.)

- Match the string “100%”

  LIKE ‘100 \%’ **ESCAPE** '\'

  in that above we use backslash (\) as the escape character
- Patterns are case sensitive.
- Pattern matching examples:
  - ‘Intro%’ matches any string beginning with “Intro”.
  - ‘%Comp%’ matches any string containing “Comp” as a substring.
  - ‘_ _ _’ matches any string of exactly three characters.
  - ‘_ _ _ %’ matches any string of at least three characters.

Range Queries

- SQL includes a **BETWEEN** comparison operator
- Example: Find the names of all instructors with salary between $90,000 and $100,000 (that is, ≥ $90,000 and ≤ $100,000)

```
SELECT name
FROM instructor
WHERE salary BETWEEN 90000 AND 100000
```

Tuple Comparison

SELECT *name, course_id*
FROM *instructor, teaches*
WHERE *(instructor.ID, dept_name) = (teaches.ID, 'Biology')*

Set Operations

- Find courses that ran in Fall 2009 or in Spring 2010

  (SELECT *course_id* FROM *section* WHERE *sem = 'Fall'* AND *year = 2009)*
  UNION
  (SELECT *course_id* FROM *section* WHERE *sem = 'Spring'* AND *year = 2010)*
- Find courses that ran in Fall 2009 or in Spring 2010

  (SELECT *course_id* FROM *section* WHERE *sem = 'Fall'* AND *year = 2009)*
  INTERSECT
  (SELECT *course_id* FROM *section* WHERE *sem = 'Spring'* AND *year = 2010)*
- Find courses that ran in Fall 2009 or in Spring 2010

  (SELECT *course_id* FROM *section* WHERE *sem = 'Fall'* AND *year = 2009)*
  EXCEPT
  (SELECT *course_id* FROM *section* WHERE *sem = 'Spring'* AND *year = 2010)*

Null Values

- It is possible for tuples to have a null value, denoted by *NULL*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *NULL* is *NULL*
  - Example: 5 + *NULL* returns NULL
- The predicate **IS NULL** can be used to check for null values.
  - Example: Find all instructors whose salary is null.

```
SELECT name
FROM  instructor
WHERE salary IS NULL
```

Null Values and Three Valued Logic

- Three values – *true*, *false*, *unknown*
- Any comparison with *null* returns *unknown*
  - Example*: 5 < null or null <> null or null = null*
- Three-valued logic using the value *unknown*:
  - OR: (*unknown* **OR** *true*) = *true*,
    (*unknown* **OR** *false*) = *unknown*
    (*unknown* **OR** *unknown) = unknown*
  - AND: *(true* **AND** *unknown) = unknown,*
    (false **AND** *unknown) = false,*
    (unknown **AND** *unknown) = unknown*
  - NOT*: (***NOT** *unknown) = unknown*
  - "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of **WHERE** clause predicate is treated as *false* if it evaluates to *unknown*

The IN Operator

- <v> **IN** <S> evaluates to true if the value v matches one of the values in S.
- It can be used to replace a sequence of conditions connected by **OR**
- **Example:**

```
SELECT name
FROM instructor
WHERE dept_name IN ('Comp. Sci.', Math.', 'Chem.');
```

This Query is equivalent to:

```
SELECT name
FROM instructor
WHERE dept_name = 'Comp. Sci.' OR dept_name = Math.' OR dept_name =
'Chem.'
```

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

Aggregate Functions (Cont.)

- Find the average salary of instructors in the Computer Science department

```
SELECT AVG (salary)
FROM instructor
WHERE dept_name= 'Comp. Sci.';
```

- Find the total number of instructors who teach a course in the Spring 2010 semester

```
SELECT COUNT (DISTINCT ID)
FROM teaches
WHERE semester = 'Spring' AND year = 2010;
```

- Find the number of tuples in the *course* relation

```
SELECT COUNT (*)
FROM course;
```

Aggregate Functions (Cont.)

- Find the average salary of instructors in each department

```
SELECT dept_name, AVG (salary) AS avg_salary
FROM instructor
GROUP BY dept_name;
```

ID	name	dept_name	salary
22322	Einstein	Physics	95000
33452	Gold	Physics	87000
21212	Wu	Finance	90000
10101	Brandt	Comp. Sci.	82000
43521	Katz	Comp. Sci.	75000
98531	Kim	Biology	78000
58763	Crick	Elec. Eng.	80000
52187	Mozart	History	65000
32343	El Said	History	86000

The query result

dept_name	avg_salary
Physics	91000
Finance	90000
Comp. Sci.	78500
Biology	78000
Elec. Eng.	80000
History	75500

Aggregate Functions (Cont.)

- Find the average salary of instructors in each department which has average salary greater than 80000 – use HAVING because WHERE cannot be used with aggregate functions

  SELECT *dept_name*, AVG (*salary*) AS *avg_salary*
  FROM *instructor* GROUP BY *dept_name*;
  HAVING avg_salary > 80000

The query result

dept_name	avg_salary
Physics	91000
Finance	90000

ID	name	dept_name	salary
22322	Einstein	Physics	95000
33452	Gold	Physics	87000
21212	Wu	Finance	90000
10101	Brandt	Comp. Sci.	82000
43521	Katz	Comp. Sci.	75000
98531	Kim	Biology	78000
58763	Crick	Elec. Eng.	80000
52187	Mozart	History	65000
32343	El Said	History	86000

Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **SELECT-FROM-WHERE** expression that is nested within another query.
- The nesting can be done in the following SQL query

SELECT $A_1, A_2, \ldots, A_n$
FROM $r_1, r_2, \ldots, r_n$
WHERE P

as follows:

- A_i can be replaced be a subquery that generates a single value.
- r_i can be replaced by any valid subquery
- P can be replaced with an expression of the form:

  B <operation> (subquery)

Subqueries – Examples

- Find courses offered in Fall 2009 and in Spring 2010

```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' AND year= 2009 AND
  course_id IN (SELECT course_id
                FROM section
                WHERE semester = 'Spring' AND year= 2010);
```

- Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

```
SELECT dept_name, avg_salary
FROM (SELECT dept_name, AVG (salary) AS avg_salary
        FROM instructor
        GROUP BY dept_name)
WHERE avg_salary > 42000;
```

Subqueries – Examples (Cont.)

- List all departments along with the number of instructors in each department

```
SELECT dept_name,
        (SELECT COUNT(*)
           FROM instructor
          WHERE department.dept_name = instructor.dept_name)
        AS num_instructors
FROM department;
```

- Runtime error if subquery returns more than one result tuple
- Note that: subqueries are parenthesized SELECT-FROM-WHERE statements

Demo

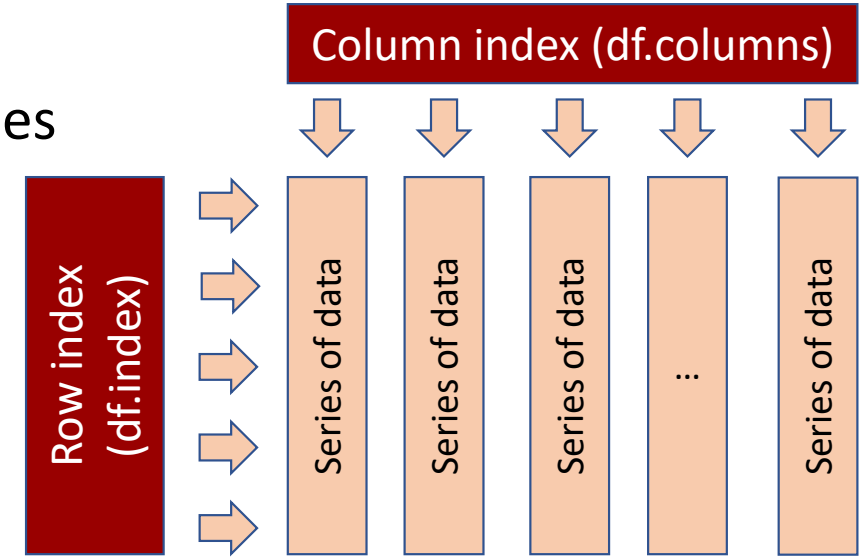
- https://www.w3schools.com/sql/trysql.asp?filename=trysql_op_in

Data Extraction in Python

Utrecht University

Pandas Dataframes

- The most popular way to handle data tables in Python is using Pandas dataframes
- DataFrame: a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.)
- Has columns and rows indexes
- Columns are made up of pandas series



Creating DataFrame

In [1]:
```
import pandas as pd
data = {'State': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
 'Year': [2000, 2001, 2002, 2001, 2002, 2003],
 'Population': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
df = pd.DataFrame(data)
```

In [2]:
```
df
```

Out[2]:

	State	Year	Population
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

- Similarly: you can use the following code

```
import pandas as pd
data = [['Ohio', 2000, 1.5], ['Ohio', 2001, 1.7],
        ['Ohio', 2002, 3.6], ['Nevada', 2001, 2.4],
        ['Nevada', 2002, 2.9], ['Nevada', 2003, 3.2]]
cols = ['State', 'Year', 'Population']
df = pd.DataFrame(data, columns = cols)
```

Load DataFrame from CSV Files

- The simplest way is:

```
df = pd.read_csv('file.csv’)    # often works
```

- More options can be added when loading a csv file into a dataframe

```
df = pd.read_csv('movies.csv', header=0,
    index_col=0, quotechar='"',sep=",",
    na_values = ['na', '-', '.', “])
```

- More options can be found in Pandas documentation
- Remeber to import the pandas library as pd

Load DataFrame from EXCEL Files

- Each Excel sheet in a Python dictionary

```
workbook = pd.ExcelFile('file.xlsx')
dictionary = {}
for sheet_name in workbook.sheet_names:
    df = workbook.parse(sheet_name)
    dictionary[sheet_name] = df
```

- The parse() method takes many arguments like read_csv().
- Refer to the pandas documentation

Load DataFrame from MySQL Database

- Create Connector/Python to MySQL

```
from mysql.connector import (connection)
cnx = connection.MySQLConnection(user='user_name',
        password='password', host='127.0.0.1', database='university')
cursor = cnx.cursor()
```

- Create SQL query as a string and send the query to the DBMS

```
query = ("SELECT ID, name, tot_cred   FROM student WHERE tot_cred > 24")
cursor.execute(query)
```

- Process the results of the query and close the connection

```
df = pd.DataFrame(cursor, columns = ["ID", "name", "tot_cred"])
cursor.close()
cnx.close()
```

Load DataFrame from PostgreSQL Database

- Create Connector/Python to PostgreSQL using psycopg2 library

```
import psycopg2
conn = psycopg2.connect(database='db_name', user='user_name',
                        password='password', host='127.0.0.1', port=5432)
cursor = conn.cursor()
```

- Create SQL query as a string and send the query to the DBMS

```
query = ("SELECT ID, name, tot_cred   FROM student WHERE tot_cred > 24")
cursor.execute(query)
```

- Process the results of the query and close the connection

```
df = pd.DataFrame(cursor, columns = ["ID", "name", "tot_cred"])
cursor.close()
cnx.close()
```

Working with Dataframes

- Consider the movies dataset extracted from imdb dataset
- Start by reading the csv file

```
df = pd.read_csv(filepath_or_buffer = 'movies.csv', delimiter=',',
      doublequote=True, quotechar='"',na_values = ['na', '-', '.', ''],
      quoting=csv.QUOTE_ALL, encoding = "ISO-8859-1")
```

- Extract sub-table of the dataframe

```
df.info()                    # index & data types
n = 4
dfh = df.head(n)             # get first n rows
dft = df.tail(n)             # get last n rows
dfs = df.describe()          # summary stats cols
top_left_corner_df = df.iloc[:5, :5]
```

Extracting Data from Dataframes

- Extarct row number 0

```
row1 = df.iloc[0,:]     #You may ignore adding the :
row1 = df.iloc[0]
```

- Extract the column with the names of directors

```
df.director_name     #  OR
df["director_name"]
```

Extracting Data from Dataframes (Cont.)

- Extract set of rows (corresponds to selection in relational algebra)

```
Rows_set1 = df.iloc[[5:10], ]          # Extracts rows 5,6,7,8, and 9
Rows_set2 = df.iloc[[5,6,8,10], ]     # Extracts rows 5,6,8, and 10
```

- Extract set of columns (corresponds to projection in relational algebra)

```
cols_set1 = df[df.columns[5:10]][:]          # Extracts columns 5,6,7,8, and 9
cols_set2 = df[df.columns[[5,7,9]]][:]     # Extracts rows 5,6,8, and 10
col_set3 = df[['actor_3_facebook_likes', 'actor_1_facebook_likes', 'content_rating']]
```

- Note that: df.columns is a vector that contains the attributes'names

Extracting Data from Dataframes (Cont.)

- Extract set of rows with a condition

```
df.loc[df['content_rating'] == 'PG-13', ['actor_1_facebook_likes',
        'actor_3_facebook_likes', 'budget']]
```

- You can do the same thing using iloc

```
df.iloc[(df['content_rating'] == 'PG-13').values, [1, 3]]
```

- Note that: iloc requires numerical values for the indexes

Extracting Data from Dataframes (Cont.)

- Extract set of rows with a condition

```
df.loc[df['content_rating'] == 'PG-13', ['actor_1_facebook_likes',
        'actor_3_facebook_likes', 'budget']]
```

- You can do the same thing using iloc

```
df.iloc[(df['content_rating'] == 'PG-13').values, [1, 3]]
```

- Note that: iloc requires numerical values for the indexes

Profiling the Dataframes

- Display number of columns

```
print(len(df.columns))
```

- Display number of rows

```
print(len(df))      # OR print(len(df[df.columns[0]])
```

- Find the number of non-null values in each column (attribute)

```
df.count()
```

Profiling the Dataframes

- Display number of distinct values in an attribute

```
for col in  df.columns:
    print(col, ' has (', len(df[col].unique()), ') unique values')
```

- Display the data type of each attribute

```
dataTypeSeries = df.dtypes
for col_idx in range(len(df.columns)):
            print(df.columns[col_idx], 'has type (', dataTypeSeries[col_idx], ')')
```

Profiling the Dataframes – Aggregate Queries

- Find max, min, and average of numerical attributes

```
dataTypeSeries = df.dtypes
for col_idx in range(len(df.columns)):
    if (not (dataTypeSeries[col_idx] == 'object')):
        print(df.columns[col_idx], 'has Min = ', df[df.columns[col_idx]].min(),
            'Max = ', df[df.columns[col_idx]].max(),
            'Avergare = ', df[df.columns[col_idx]].mean())
```

Set Operations on Dataframes

- Assume the following dataframes

```
dd1 = pd.DataFrame( { 'id': ['1', '2', '3', '4', '5'], 'Feature1': ['A', 'C', 'E', 'G', 'I'],
        'Feature2': ['B', 'D', 'F', 'H', 'J']})
```

```
dd2 = pd.DataFrame( { 'id': ['1', '2', '6', '7', '8'], 'Feature1': ['A', 'C', 'O', 'Q', 'S'],
        'Feature2': ['B', 'D', 'P', 'R', 'T']})
```

- The *concat* function concatenates the dataframes allowing repetition

```
union_df = pd.concat([dd1, dd2])                    # concatenate row-wise
union_df = pd.concat([dd1, dd2], axis = 1)          # concatenate column-wise
```

Join Operation on Dataframes

- The *merge* function joins dataframes on selected attribute

```
df_merge_col = pd.merge(dd1, dd2, on='id')
```

- If the joining attribute has different names in both dataframes

```
df_merge_col = pd.merge(dd1, dd2, left_on='att_dd1', right_on = 'att_dd2')
```

Demo

- Examples of data extraction and profiling in Python
- We will use jupyter notebook

Further Reading Material & Exercise

- Chapters 3.4-4.1 of the Database System Concepts Book
- Chapters 5-6 and 8 of the Python for Data Analysis Book