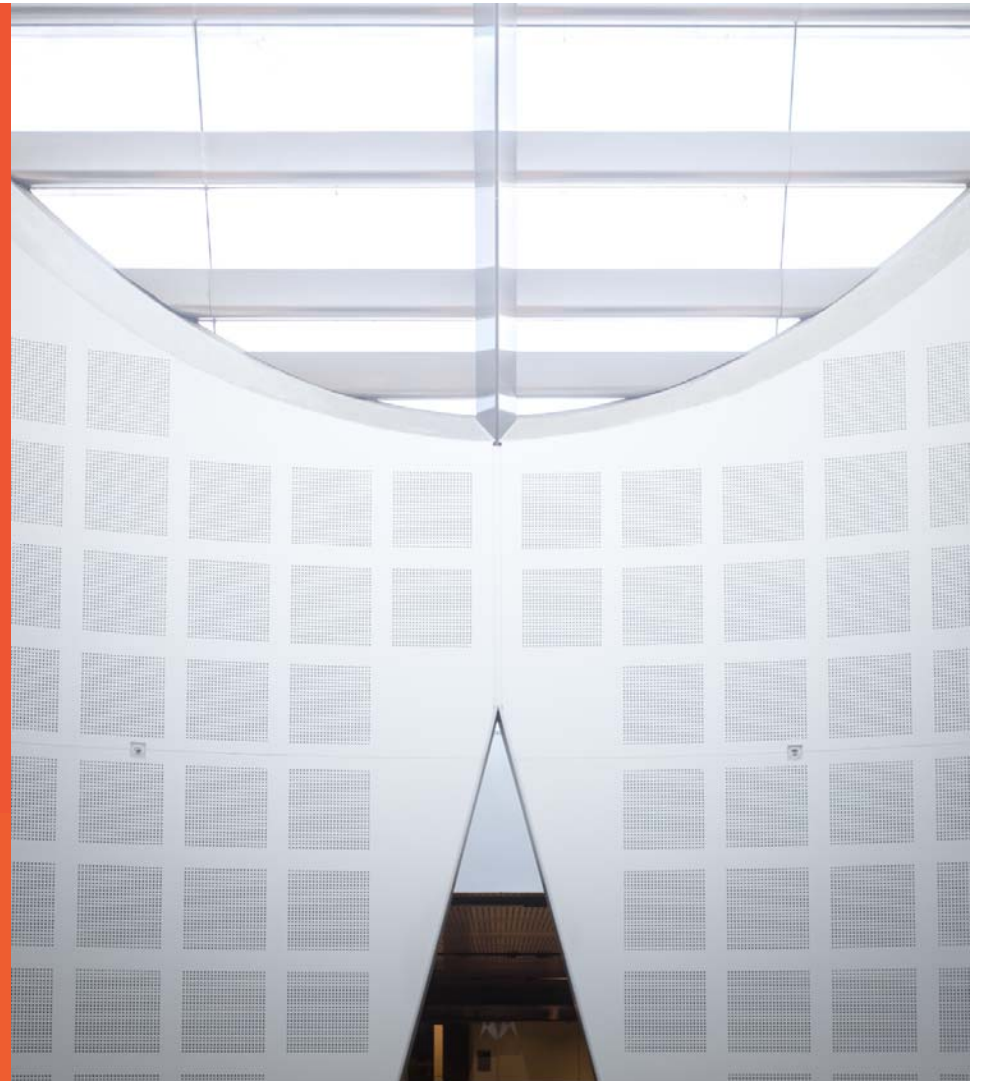


DATA2001: Data Science: Big Data and Data Diversity

W4: Declarative data analysis with SQL

Presented by

Dr. Matloob Khushi
School of IT



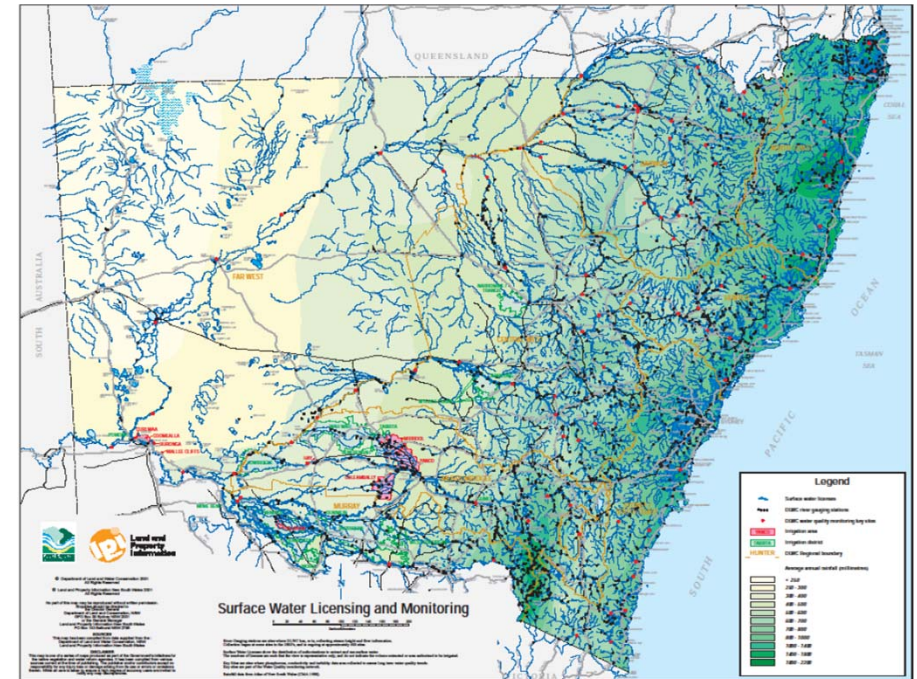
Overview of last week

Where do we get data from?

- You or your organization might have it already, or a colleagues provides you access to data.
 - Typical exchange formats: CSV, Excel, XML/JSON
- Or: Download from an online data server
 - Still typically in CSV or Excel etc, but now problems with meta-data
- Or: Scrap the web yourself or use APIs of resources
 - Cf. textbook, chapter 9

Murray River Basin Data Set


- Water measurements:
 - automatic monitoring stations
 - that are distributed over a larger area
 - (say the Murray River catchment basin)
 - that periodically send their measured values to a central authority:
 - Time-series data of water level, water flow, water temperature, salinity (via measuring electric conductivity) or other hydraulic properties



Relational Databases


- **Informal Definition:**
A **relation** is a named, two-dimensional table of data
 - Table consists of rows (record) and columns (attribute or field)
- Example:

Attributes (also: columns, fields)



<i>Student</i>				
<u>sid</u>	name	login	gender	address
5312666	Jones	ajon1121@cs	m	123 Main St
5366668	Smith	smith@mail	m	45 George
5309650	Jin	ojin4536@it	f	19 City Rd

Tuples
(rows,
records)



This Week: RDBMS + Python

- Today's goal is to store the data available as CSV files in a relational database using Python
- **Relational data model** is the most widely used model today
 - Main concept: **relation**, basically a table with rows and columns
- This sounds like a spreadsheet, but as we have seen last week, it has some differences

Some Remarks

- Not all tables qualify as a relation:
 - Every relation must have a unique name.
 - Attributes (columns) in tables must have unique names.
 - => The order of the columns is irrelevant.
 - All tuples in a relation have the same structure; constructed from the same set of attributes
 - Every attribute value is atomic (not multivalued, not composite).
 - Every row is unique
(can't have two rows with exactly the same values for all their fields)
 - The order of the rows is immaterial

SQL (Structured Query Language)

SQL – The Structured Query Language

- SQL is the standard declarative query language for RDBMS
 - Describing what data we are interested in, but not how to retrieve it.
- Supported commands from roughly two categories:
 - **DDL** (Data Definition Language)
 - Create, drop, or alter the relation schema
 - Example:
CREATE TABLE *name* (*list_of_columns*)
 - **DML** (Data Manipulation Language)
 - for retrieval of information also called **query language**
 - **INSERT, DELETE, UPDATE**
 - **SELECT ... FROM ... WHERE**

SQL DML Statements

- Insertion of new data into a table / relation
 - **Syntax:**
INSERT INTO *table* [“(”*list-of-columns*“)”] **VALUES** “(“ *list-of-expression* “)”
 - Example:
INSERT INTO Students (sid, name) VALUES (53688, 'Smith')
- Updating of tuples in a table / relation
 - **Syntax:**
UPDATE *table* **SET** *column*“=“*expression* {“,”*column*“=“*expression*}
[**WHERE** *search_condition*]
 - Example: **UPDATE students**
SET gpa = gpa - 0.1
WHERE gpa >= 3.3
- Deleting of tuples from a table / relation
 - **Syntax:**
DELETE FROM *table* [**WHERE** *search_condition*]
 - Example:
DELETE FROM Students WHERE name = 'Smith'

Table Constraints and Relational Keys

- When creating a table, we can also specify **Integrity Constraints** for columns
 - eg. domain types per attribute, or **NULL / NOT NULL** constraints
- **Primary key:** unique, minimal identifier of a relation.
 - Examples include employee numbers, social security numbers, etc. This is how we can guarantee that all rows are unique.
- **Foreign keys** are identifiers that enable a dependent relation (on the many side of a relationship) to refer to its parent relation (on the one side of the relationship)
 - Must refer to a candidate key of the parent relation
 - Like a 'logical pointer'
- Keys can be **simple** (single attribute) or **composite** (multiple attributes)

Example: Relational Keys

Primary key identifies each tuple of a relation.

Composite Primary Key consisting of more than one attribute.

<i>Student</i>	
<u>sid</u>	name
31013	John

<i>Enroll</i>		
<u>sid</u>	<u>ucode</u>	grade
31013	I2120	CR

<i>Units_of_study</i>		
<u>ucode</u>	title	credit_pts
I2120	DB Intro	4

Foreign key is a (set of) attribute(s) in one relation that 'refers' to a tuple in another relation (like a 'logical pointer').

SQL Domain Constraints

- SQL supports various domain constraints to restrict attribute to valid domains
 - NULL / NOT NULL whether an attribute is allowed to become *NULL* (unknown)
 - DEFAULT to specify a default value
 - CHECK(*condition*) a Boolean *condition* that must hold for every tuple in the db instance

Example:

```
CREATE TABLE Student
(
    sid          INTEGER          PRIMARY KEY,
    name         VARCHAR(20)     NOT NULL,
    gender       CHAR            CHECK (gender IN ('M','F','T')),
    birthday     DATE            NULL,
    country      VARCHAR(20),
    level        INTEGER         DEFAULT 1 CHECK (level BETWEEN 1 and 5)
);
```

Combining data from multiple tables

For example, the following query lists all stations belong to **Victoria Government**

set search_path to waterwaysdata;

```
SELECT *  
FROM Stations, Organisations  
WHERE Stations.owner = Organisations.code  
AND Organisations.name = 'Victoria Government';
```

All tables accessed by the query are listed in the **FROM** clause, separated by comma.

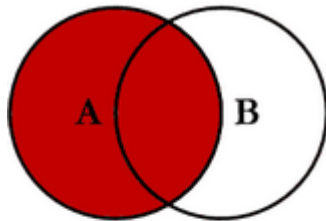
Natural Joins

A *natural join* between two tables combines those rows whose values agree in all common attributes, ie. those that are named the same. These common columns are also shown only once in the result.

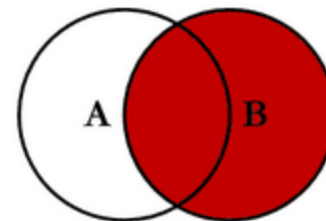
```
SELECT *  
FROM measurements NATURAL JOIN sensors;
```

If there is no common attribute in two tables; the query produces a *cross join*, also called the *cross product* of both tables (undesired result).

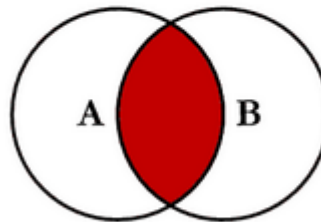
SQL JOINS



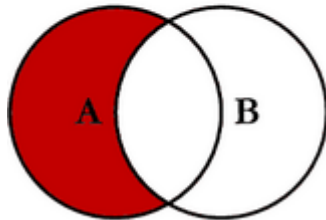
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



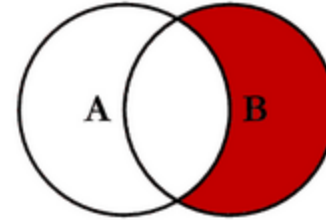
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



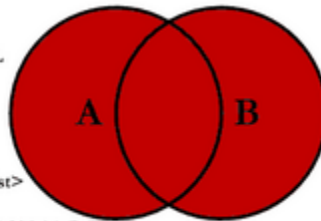
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



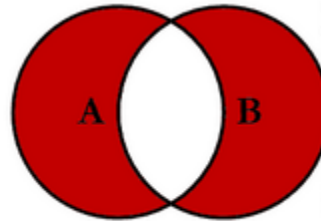
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

© C.L. Moffatt, 2008

SQL Subqueries

- A subquery is a query within a query. You can create subqueries within your SQL statements. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

```
SELECT * FROM measurement  
WHERE sensor in ( SELECT sensor FROM sensor );
```

```
select * from measurement where sensor='temp' and  
value > ( select AVG(value) from measurement where sensor='temp' );
```

DB Creation / Data Loading

Data Storing

- Data is ready?
 - We have analysed our given data set
 - Cleaned it
 - Transformed it and created a corresponding relational database
- Next, we want to store the given data in our database.
- Main approaches:
 1. Command line tools
 2. Python loader
 3. (Combination of Python loader and stored procedures)

Approach 1: PSQL Data Loader

- Postgresql offers a command to load data directly from a CSV file into a database table
 - `\COPY tablename FROM filename CSV [HEADER] [NULL '...']`
 - Many further options
 - Try `\help COPY`
- Pros:
 - Relatively fast and straight-forward
 - No programming needed
- Cons:
 - Only 1:1 mapping of CSV to tables; no data cleaning or transformation
 - Stops at the first error...

Approach 2: Python Loading Code

- Example: Creating a table and loading some data

```
# 1st: login to database|
conn = pgconnect()

# 2nd: ensure that the schema is in place
organisation_schema = """CREATE TABLE IF NOT EXISTS Organisation (
                           code VARCHAR(20) PRIMARY KEY,
                           orgName   VARCHAR(150)
                           )"""
pgexec (conn, organisation_schema, None, "Create Table Organisation")

# 3rd: load data
# IMPORTANT: make sure the header line of CSV is without spaces!
insert_stmt = """INSERT INTO Organisation(code,orgName)
                  VALUES (%(Code)s, %(Organisation)s)"""
for row in data_organisations:
    pgexec (conn, insert_stmt, row, "row inserted")
```

- Pros: Full flexibility; data cleaning and transformation possible
- Cons: Has to be hand-coded for each case

Exercise : Some initial issues to be solved

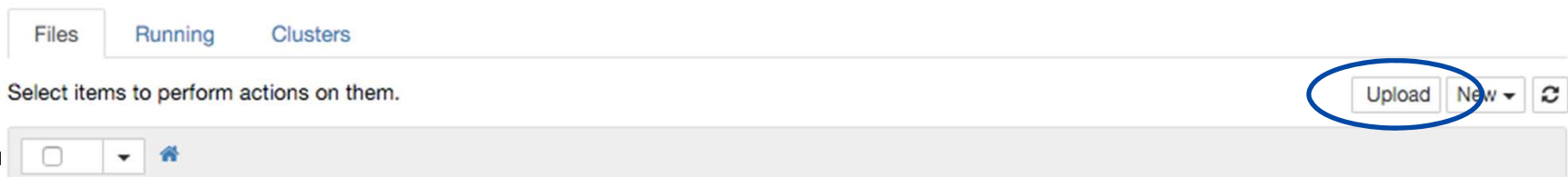
- Our first attempt is for just a 1:1 mapping
 - Still: separate tables need to go to separate database relations
 - Note: CSV headers are not allowed to contain spaces or '
- Connection to Postgresql with **psql** shell tool
- Create 1:1 mapping tables in SQL
- Load CSV directly to SQL tables
 - We first try COPY command from psql
<http://www.postgresql.org/docs/current/interactive/sql-copy.html>

Exercise : Data Loading with DB Loader

- Download data and notebook from **Canvas**
 - CSV data files:
 - Measurements.csv
 - Organisations.csv
 - Sensors.csv
 - Stations.csv
 - Jupyter Notebook
- Upload all those files to Jupyter server
- Run notebook and do Exercise 1 for db creation and data loading with **psql**

Important:

Make sure to use the correct names including the '.csv' file extension



Reminder: Jupyter server locations

1. Login to one of these Jupyter Servers

- <https://ucpu1.ug.it.usyd.edu.au/>
- <https://soit-ucpu-pro-X.ucc.usyd.edu.au> (**X** = 1 or 3)
- Log in with unikey as username and password
- Both servers are linked to your ICT shared folder here at Usyd
 - you need to have logged-in to one of the lab machines in the SIT building first to have this shared folder created

2. Login to our central postgresql: soit-db-pro-2.ucc.usyd.edu.au Your y18s1d2001_unikey with your SID as password

Database Loading with Python

Issues with DB Loaders

- DB Loading tools such as **psql**
 - good for administration of server
 - good for bulk-loading of exported data in clean csv format (db export)
 - needs terminal access (both an advantage and a disadvantage)
 - has its limitations if format and structure of data files do not match the actual database schema
 - => cleaning and transformation of data needed
- Could we do so in a programming language such as Python?

Accessing PostgreSQL from Python: psycopg2

- First, we need to import the **psycopg2** module, then connect to Postgresql
- Note: You need obviously to provide you own login name
 - Username and password are prepared by us as your unikey and your SID

```
import psycopg2

def pgconnect():
    # please replace <your_unikey> and <your_SID> with your own details
    YOUR_UNIKEY = '<your_unikey>'
    YOUR_PW     = '<your_SID>'
    try:
        conn = psycopg2.connect(host='soit-db-pro-2.ucc.usyd.edu.au',
                                database='y18s1c5310_'+YOUR_UNIKEY,
                                user='y18s1c5310_'+YOUR_UNIKEY,
                                password=YOUR_PW)

        print('connected')
    except Exception as e:
        print("unable to connect to the database")
        print(e)
    return conn
```

Accessing PostgreSQL from Python: psycopg2 (cont'd)

- How to execute an SQL statement on an open connection 'conn'
 - we prepared a helper function which encapsulates all the error handling:

```
def pgexec( conn, sqlcmd, args, msg ):
    """ utility function to execute some SQL statement
        can take optional arguments to fill in (dictionary)
        error and transaction handling built-in """
    retval = False
    with conn:
        with conn.cursor() as cur:
            try:
                if args is None:
                    cur.execute(sqlcmd)
                else:
                    cur.execute(sqlcmd, args)
                print("success: " + msg)
                retval = True
            except Exception as e:
                print("db error: ")
                print(e)
    return retval
```

Accessing PostgreSQL from Python: psycopg2 (cont'd)

- Example: Creating a table and loading some data

```
# 1st: login to database|
conn = pgconnect()

# 2nd: ensure that the schema is in place
organisation_schema = """CREATE TABLE IF NOT EXISTS Organisation (
                           code VARCHAR(20) PRIMARY KEY,
                           orgName   VARCHAR(150)
                           )"""
pgexec (conn, organisation_schema, None, "Create Table Organisation")

# 3rd: load data
# IMPORTANT: make sure the header line of CSV is without spaces!
insert_stmt = """INSERT INTO Organisation(code,orgName)
                  VALUES %(Code)s, %(Organisation)s)"""
for row in data_organisations:
    pgexec (conn, insert_stmt, row, "row inserted")
```

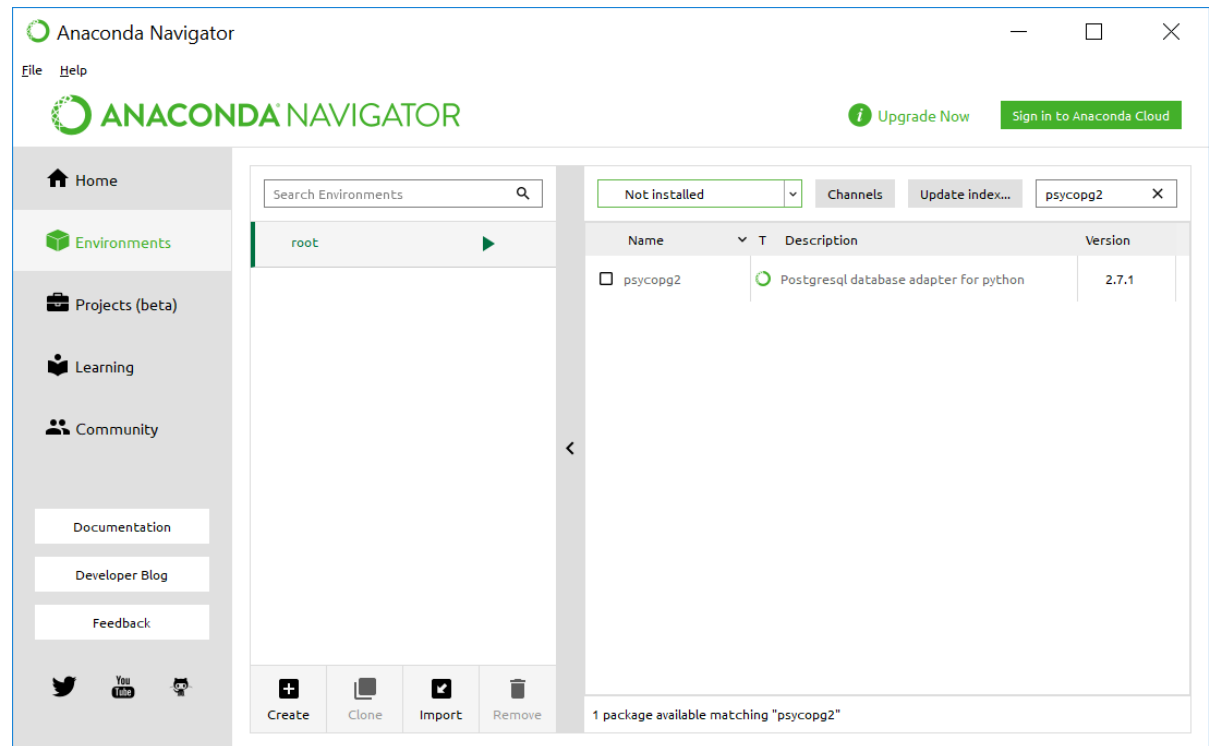
How to use psycopg2 on your private installation

- If you are using a local installation of Jupyter notebooks on your laptop, you need to install a database and the corresponding Python module first
- Tip given at end of Exercise 2 on how to do this on macOS
 - macOS includes a postgresql server already
- You should be able to access our central postgresql server from your own laptop to:

```
psql -h soit-db-pro-2.ucc.usyd.edu.au -d y18s1c5310_UNIKEY -U y18s1c5310_UNIKEY
```

Connecting to localhost database

- Install psycopg2 using Anaconda Navigator
- Change connection string in Jupyter



```
conn = psycopg2.connect(host='localhost', database='waterinfo', user='matloob', password='matloob')
```

Exercise : Data Loading with Python

- Code in Jupyter notebook
 - Load CSV data into Python
 - Helper functions for connecting and querying postgresql
 - important: **Edit your login details** in the pgconnect() function
 - Check content of Organisation table
- **Your task:** Doing the same for the ‘Measurements’ and ‘Stations’ data
 - table creation & data loading in Python
- Any other observations?
- What problems do you encounter when trying to load the table?

Transforming and Cleaning Data

Issues encountered at previous exercises

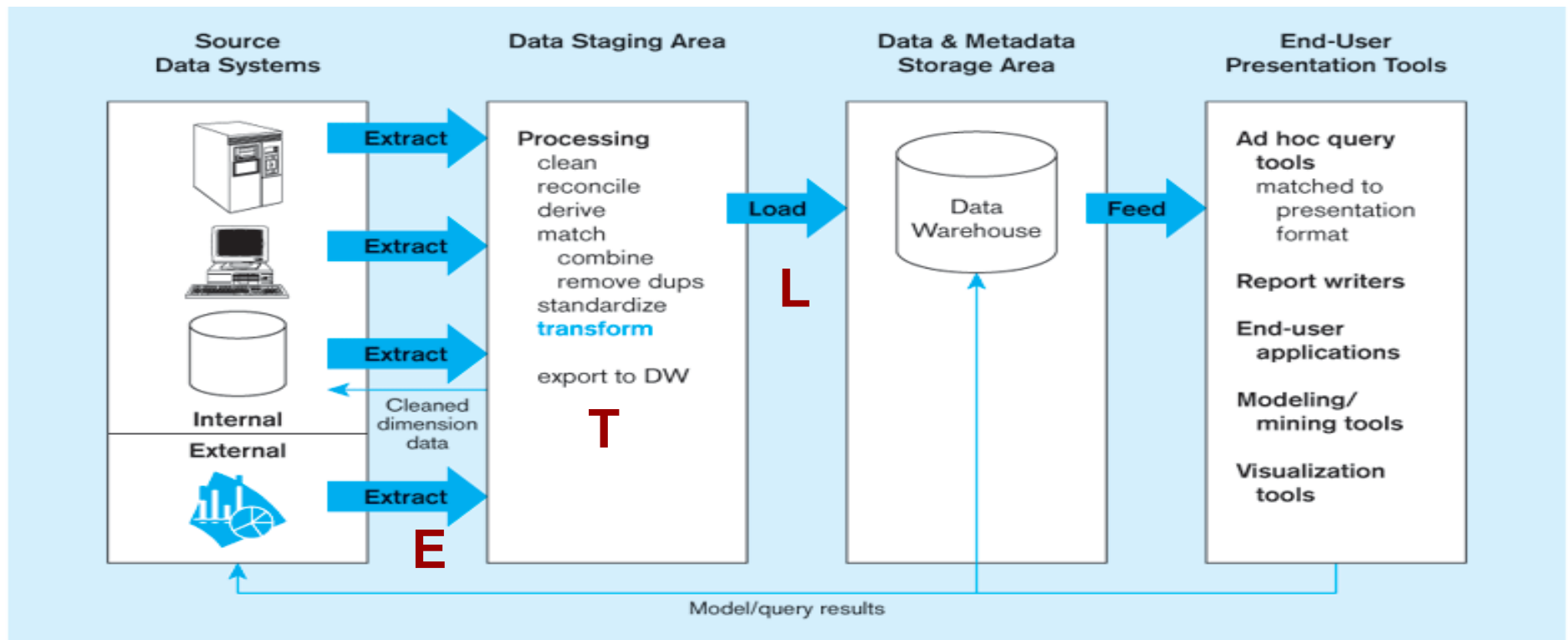
- Interpretation of data format and meta-data
- Differences in naming conventions
 - Excel headers with spaces and quotes, which both are not allowed to DBMS
- Inconsistent or missing data entries
- 'shape' of data

Cleaning and Transforming Data

- Real data is often '*dirty*'
- Important to do some data cleaning and transforming first
 - remember last week, when we had to convert strings to float
- Typical steps involved:
 - type and name conversion
 - filtering of missing or inconsistent data
 - unifying semantic data representations
 - matching of entries from different sources
- Later also:
 - **Rescaling** and optional dimensionality reduction

ETL Process

- This problem is well known from data warehousing
- **ETL Process:** Capture/Extract - Data Cleansing - Transform - Load



Data Cleaning in Python

- Yes, there are powerful ETL tools out there, but we do it for free in Python:
 - (1) type and name conversion
 - (2) filtering of missing or inconsistent data
 - (3) unifying semantic data representations
 - (4) matching of entries from different sources
- Last week's `clean()` function deals with Tasks (1) and (2)
 - `int()` creates integer objects, e.g., -1, 101
 - `float()` creates floating point object, e.g., 3.14, 2.71
 - `datetime.strptime()` creates datetime objects from strings
 - Filters missing / wrongly formatted data and replaces with default value
- For more complex cases (3) and (4), you would need special code though

A function to convert values

Use “not a number” as default value
numpy knows to ignore for some stats

```
import numpy as np
DEFAULT_VALUE = np.nan

def clean(data, column_key, convert_function, default_value):
    special_values= {} # no special values yet
    for row in data:
        old_value = row[column_key]
        new_value = default_value
        try:
            if old_value in special_values.keys():
                new_value = special_values[old_value]
            else:
                new_value = convert_function(old_value)
        except (ValueError, TypeError):
            print('Replacing {} with {} in column {}'.format(row[column_key], new_value, column_key))
            row[column_key] = new_value

# the following converts the two measurement columns to float values - or NaN
clean(data_measurements, 'Discharge', float, DEFAULT_VALUE)
clean(data_measurements, 'MeanDischarge', float, DEFAULT_VALUE)
```

Exercise : Data Cleaning

- Refer to Jupyter notebook
 - Apply cleaning steps to data from Stations.csv
 - Repeat for **Sensors.csv** and the remaining CSVs from Canvas
- Which types of data conversions do you encounter?
- How would you resolve those?
- Any other observations regarding the source data format?

Data Modeling

Two identifiers in Stations.csv file

- If we look closer at the 'Stations.csv' content, we see that it only partially fits a relational database model
 - **station identifier** is split over two attributes
 - BasinNo
 - Site
- Human interpretable \neq machine readable
- So how to proceed with a database approach?
 - \Rightarrow **OLAP: Online Analytical Processing**

Data Warehouses: Fact Tables

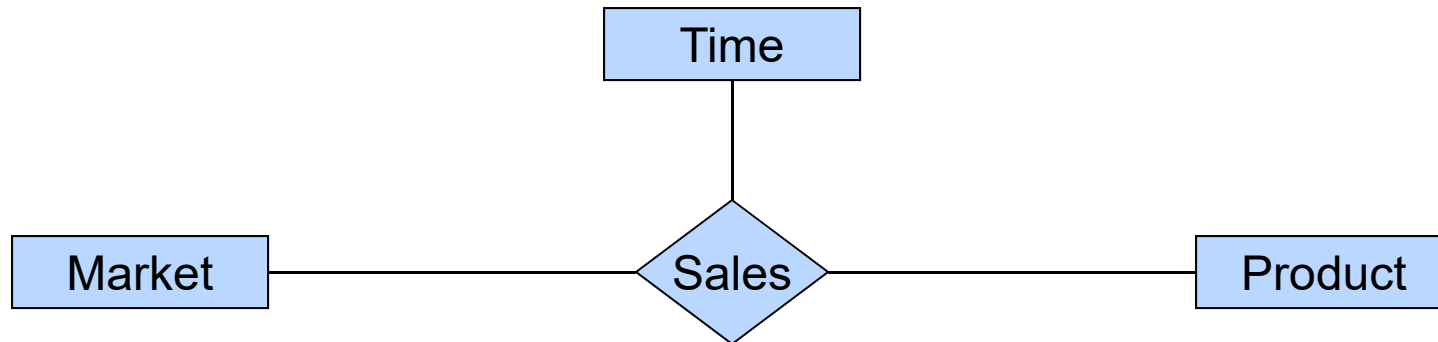
- Relational ‘data warehouse’ applications are centered around a **fact table**
 - For example, a supermarket application might be based on a table Sales (*Market_Id*, *Product_Id*, *Time_Id*, *Sales_Amt*)

market_id	product_id	time_id	sales_amt
M1	P1	T1	3000
M1	P2	T1	1000
M1	P3	T1	500
M2	P1	T1	100
M2	P2	T1	1100
M2	P3
...	...		

- The table can be viewed as *multidimensional*
 - Collection of numeric measures, which depend on a set of dimensions
 - E.g. *Market_Id*, *Product_Id*, *Time_Id* are the dimensions that represent specific supermarkets, products, and time intervals
 - *Sales_Amt* is a function of the other three

Data Warehousing: Star Schema

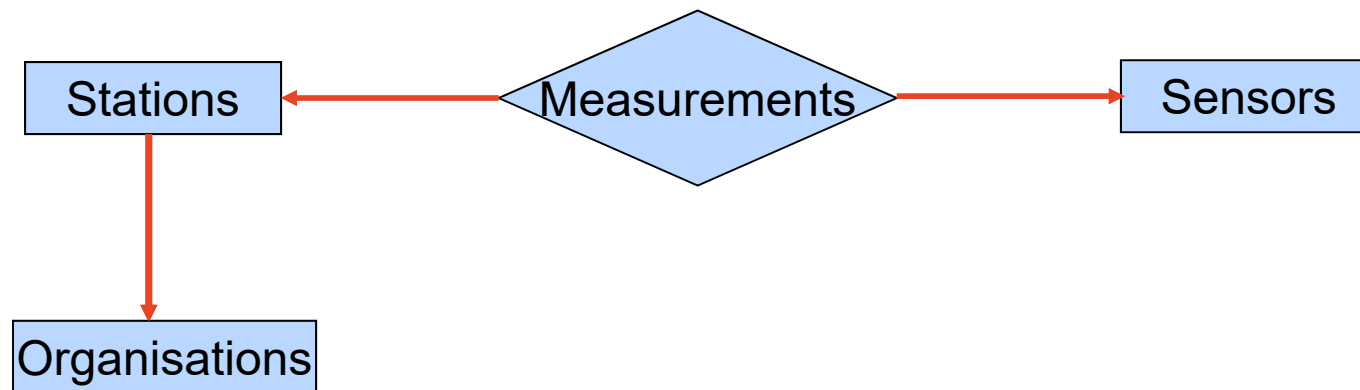
- The fact and dimension relations linked to it looks like a star;
- this is called a ***star schema***



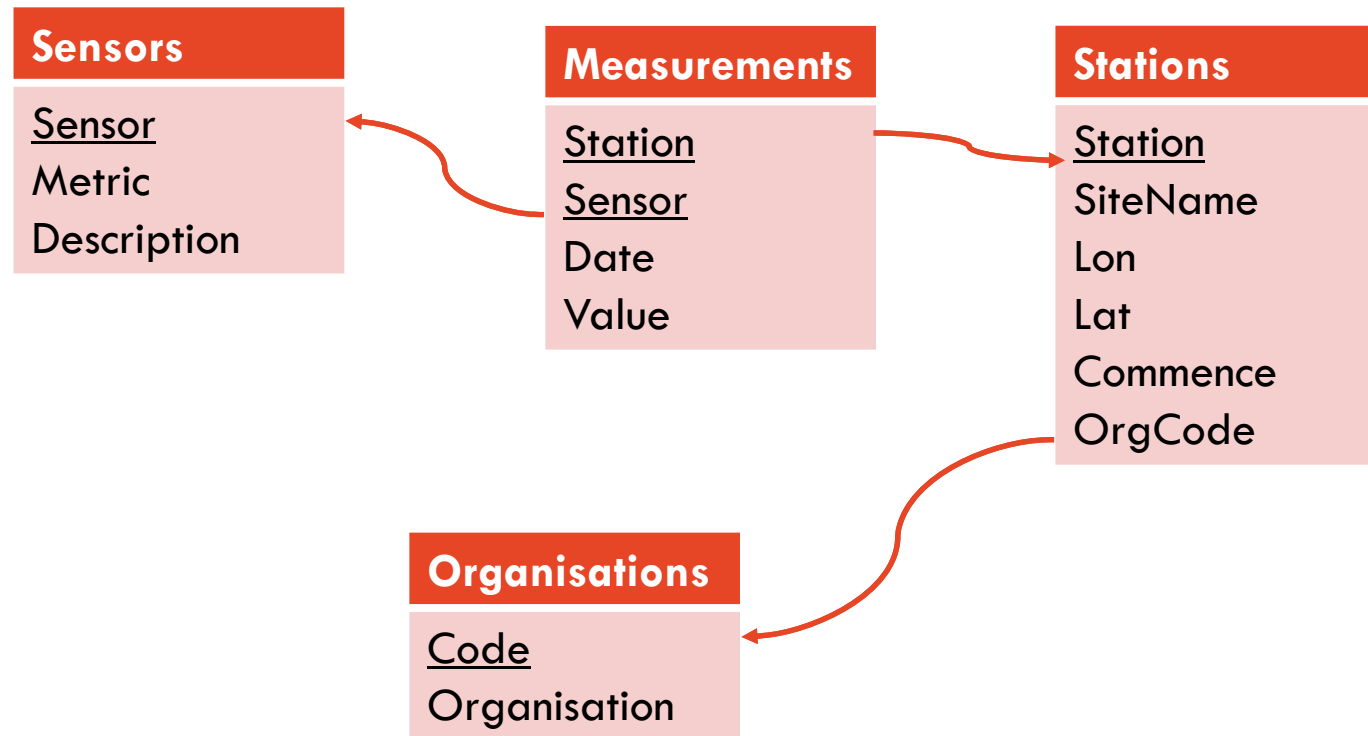
- If we map this to relations
 - 1 central fact table
 - n dimension tables with foreign key relationships from the fact table
(the fact table holds the FKs referencing the dimension tables)

Data Warehousing: Snowflake Schema

- measurements are the facts, rest describes the dimensions



Modeling our Water Data Set

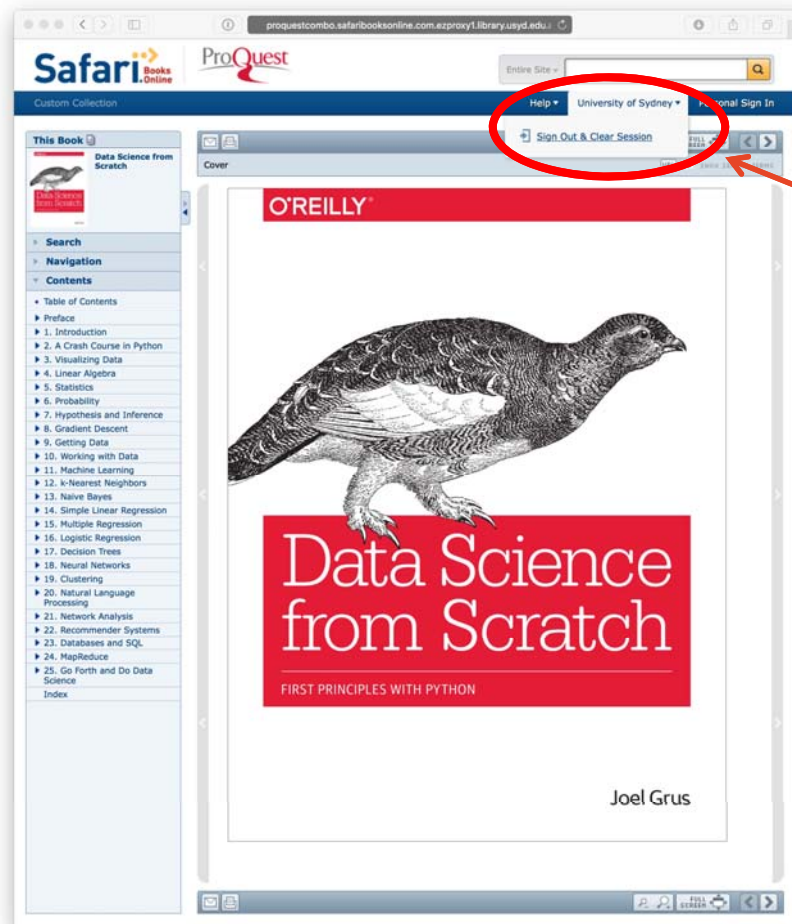


Review

Tips and Tricks

- Real data is 'dirty' – **data cleaning** and transformation essential
- Database systems are great for *shared, persistent* storage of *structured* data, and also for consistent updating ('life' data)
- But some caveats:
 - Schema-first
 - Relational model quite restrictive (1NF, no lists, collections etc)
 - Not too intuitive; 1:1 mapping from spreadsheets doomed to fail
 - Type-mismatches between programming languages and SQL
 - Needs to be installed and maintained
(though much better nowadays for SQLite and PostgreSQL)
- What's the benefit?
 - Sharing! Large data sets! Querying will give us some leverage too.

Online Book



don't forget to end session
when finished reading

Many Good Python Resources

- Hard to make recommendations given different backgrounds
- Look online, there are many free resources and example code
- A few lists:
 - <https://www.fullstackpython.com/best-python-resources.html>
 - <https://www.quora.com/Learning-Python/How-should-I-start-learning-Python-1>