# Week02 - Summary

## Databases

### Advantages of Database-approach to Data Science

- Data is managed, so **quality can be enforced by the DBMS**
  - Improved Data Sharing
    - Different users get different views of the data
    - Efficient concurrent access
  - Enforcement of Standards
    - All data access is done in the same way
  - Integrity constraints, data validation rules
  - Better Data Accessibility/Responsiveness
    - Use of standard data query language (SQL)
  - Security, Backup/Recovery, Concurrency
    - Disaster recovery is easier

- **Program-Data Independence**
  - Metadata stored in DBMS, so applications don’t need to worry about data formats
  - Data queries/updates managed by DBMS so programs don’t need to process data access routines
  - Results in:
    - Reduced application development time
    - Increased maintenance productivity

- Efficient access

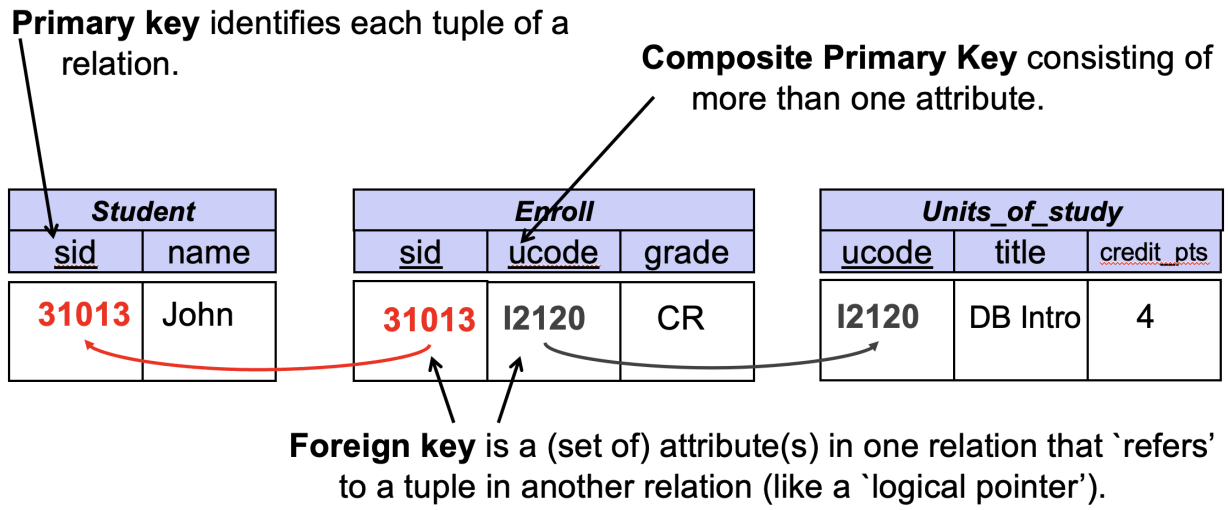
### Primary Key

- A Primary key is a unique attribute (or combination of attributes) which the database uses to identify a row in a table

### Foreign Key

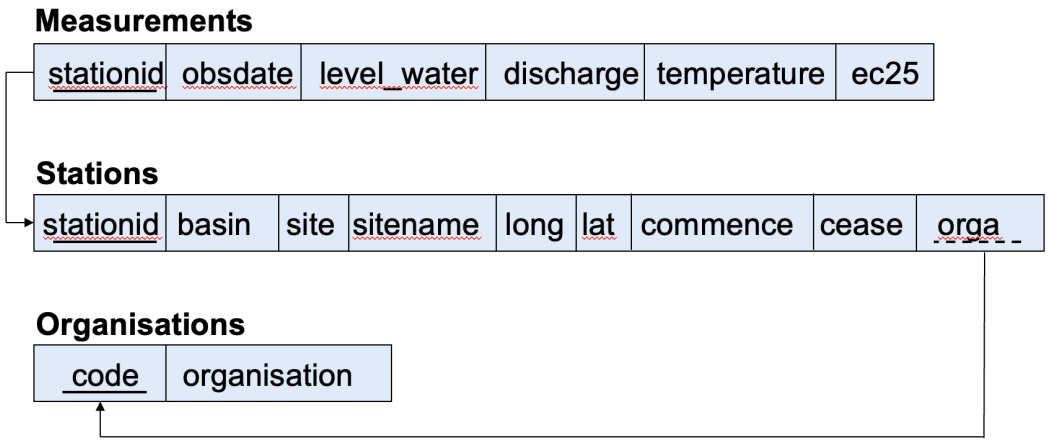
- A foreign key is defined in a second table, but it refers to the primary key or a unique key in the first table

### Example: Relational Keys



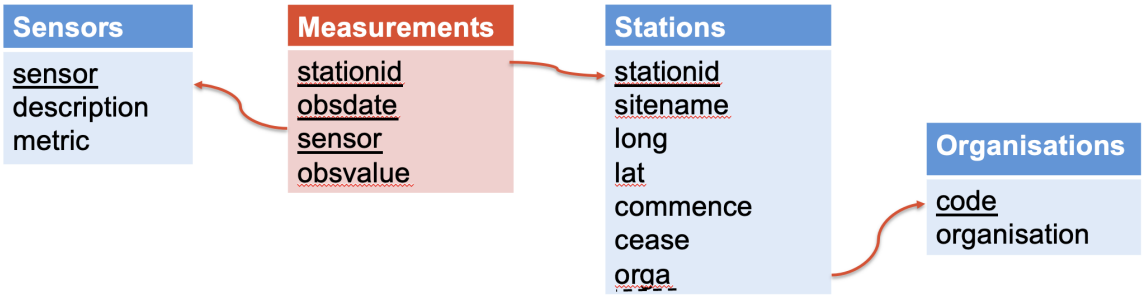

### WaterInfo Example as a relational database - Option 1

- Design Option : Straightforward 1:1 mapping
  - Because a spreadsheet is in principle a table, we can always map to 1:1
  - Some problems though: e.g. the Station ↔ Basin + Site mapping
  - A lot of NULL values for any missing measurement or if no sensor




### Modelling our water data set - Option 2

- Design Option 2: Normalised Relational Schema
  - measurements are facts, other data describes the dimensions
  - measurement entries get ‘folded’ into separate rows of the fact table
  - allows us to avoid NULLs as much as possible, but hard to read




## Data Storage with Python

### Python loading code example

```
# 1st: login to database
db, conn = pgconnect() # utility function which encapsulates db connection handling

# check existing tables
print(db.table_names())
```

```
# if you want to reset the table
conn.execute("DROP TABLE IF EXISTS Organisations")

# 2nd: ensure that the schema is in place
organisation_schema = """CREATE TABLE IF NOT EXIST Organisations (
                         code CHAR(3) PRIMARY KEY,
                         organisation VARCHAR(150)
                       )"""
conn.execute(organisation_schema)

# 3rd: load data using pandas
import pandas as pd
data = pd.read_csv('datasets/Organisations.csv')
data.to_sql("organisations", con = connn, if_exists = 'append') # Actual data storage
```

## Database Creation

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`

### Table constraints and relational keys

- When creating a table, we can specify Integrity Constraints for columns

    - e.g. 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 to a ‘logical pointer’

- Keys can be **simple** (single attribute) or **composite** (multiple attributes)

## Querying Databases with SQL

Example 1:

```
select *
from Stations;
```

Example 2:

```
select sitename, orga, commence
from Stations
where stationId = 409001;
```

Example 3:

```
select count(*)
from Stations
where orga = 'SCA';
```

## Comparison Operations

- Comparison operators in SQL: =, >, >=, <, <=, !=, <>, BETWEEN
- Comparison results can be combined using logical connectives: and, or, not
- Example 1:

```
select *
from Tablename
where (AttName1 between -90 and -50)
       and
      (AttName2 >= -45)
       and
      (AttName3 = 'H168');
```

- Example 2:

```
select *
from Stations
where siteName like 'Murray River%';
```

## Date and Time in SQL

| SQL Type | Example | Accuracy | Description |
|---|---|---|---|
| DATE | '2012-03-26' | 1 day | a date (some systems incl. time) |
| TIME | 16:12:05 | ca. 1 ms | a time, often down to nanoseconds |
| TIMESTAMP | 2012-03-26 16:12:05 | ca. 1 sec | Time at a certain date: SQL Server: DATETIME |
| INTERVAL | '5 DAY' | years - ms | a time duration |

- Comparisons

    - Normal time-order comparisons with ‘=’, ‘>’, ‘<’, ‘<=’, ‘>=’, …
- Constants
    - CURRENT_TIME db system’s current date
    - CURRENT_TIME db system’s current timestamp
- Example:

```
select *
from Measurements
where obsdate < current_date;
```

- Database systems support a variety of date/time related ops
    - Unfortunately not very standardised - a lot of slight differences
- Main Operations
    - EXTRACT (component FROM date)
        - e.e.g EXTRACT(year FROM startDate)
    - DATE string
        - e.g. DATE ‘2012-03-01’
        - Some systems allow templates on how to interpret string
    - +/- INTERVAL:
        - e.g. ‘2012-04-01’ + INTERVAL ‘36 HOUR’
- Many more → check database system’s manual

### NULL Values and three valued logic

- Any comparison with null returns unknown
    - e.g. 5 < null or null <> null or null = null

| a | b | a = b | a AND b | a OR b | NOT a | a IS NULL |
|---|---|---|---|---|---|---|
| true | true | true | true | true | false | false |
| true | false | false | false | true | false | false |
| false | true | false | false | true | true | false |
| false | false | true | false | false | true | false |
| true | NULL | unknown | unknown | true | false | false |
| false | NULL | unknown | false | unknown | true | false |
| NULL | true | unknown | unknown | true | unknown | true |
| NULL | false | unknown | false | unknown | unknown | true |
| NULL | NULL | unknown | unknown | unknown | unknown | true |

- You cannot check NULL values using equality in a database

- While NULL is very handy for representing incomplete or unknown data, NULLs and the resulting 3-valueed logic produce many corner cases in SQL:

- Result of where clause predicate is treated as false if it evaluates to unknown
  - Example: select sid from enrolled where grade = ‘DI’
    - This ignores all students without a grade so far
    - If you are interested in students without a grade, need to use the IS NULL predicate

- Aggregate functions also ignore NULLs

```
SELECT COUNT(*), COUNT(grade)
FROM Students -- can produce two different counts...
```

- Always think about potential NULL values when formulating a database query

## Analysing Data from Multiplying Tables

### JOIN: Combining Data from Multiple Tables

```
set search_path to WaterInfo;

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

### SQL Join Operators

- SQL offers several join operators which are often used in “FROM” clause to directly combine tables with matching
  - R natural join S
  - R [inner] join S on <join condition>
  - R [inner] join S using (<list of attributes>)
- E.g.
  - List of all details of the first three measurements including station details

```
SELECT *
FROM Measurements
  JOIN Stations USING (stationid)
LIMIT 3;
```

  - Which measurements were taken at station 409204 in 2016?

```
SELECT *
FROM Measurements INNER JOIN Stations USING (stationid)
WHERE stationid = 409204 AND extract(year from obsdate
= 2016;

-- Same query using JOIN operator
SELECT *
FROM Stations
  JOIN Organisations ON (orga = code)
WHERE Organisations.name = 'Victoria Government';
```

### Natural Joins

```
SELECT obsdate, obsvalue, metric
FROM Measurements NATURAL JOIN Sensors;
```

- Careful - two major pitfalls:
  - Matches any common attributes - whatever has the same name, must have the same value; so if, e.g., two tables have both a name attribute, this would have to match too, whether wanted or not
  - If there are no common attributes, it computes cross-product of two tables, which might look to you like a valid result, but is typically not what you want…
  - Hence: if in doubt, use `INNER JOIN` and formulate the join condition explicitly

### SQL Subqueries

```
SELECT *
FROM Measurements s
WHERE sensor IN (SELECT sensor FROM Sensors);

SELECT *
FROM Measurements
WHERE sensor = 'temp' and obsvalue >
                      (SELECT AVG(obsvalue)
                       FROM Measurements
                       WHERE sensor = 'temp');
```

## Summarising Data with SQL

### SQL Aggregate Functions

| SQL Aggregate Function | Meaning |
|---|---|
| COUNT(*attr*) ; COUNT(*) | Number of *Not-null-attr ; or* of all values |
| MIN(*attr*) | Minimum value of *attr* |
| MAX(*attr*) | Maximum value of *attr* |
| AVG(*attr*) | Average value of *attr* (arithmetic mean) |
| MODE() WITHIN GROUP (ORDER BY *attr*) | mode function over *attr* |
| PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY *attr*) | median of the *attr* values |
| … | … |

Example:

```
SELECT AVG(obsvalue)
FROM Measurements
WHERE stationid = 409001 AND sensor = 'level';
```

## SQL Grouping

Example: Find the average temperature measured at each day

**Measurements**

| obsdate | temp |
|---|---|
| 20 Mar 2019 | 26 C |
| 20 Mar 2019 | 24 C |
| 21 Mar 2019 | 28 C |

```
SELECT obsdate, AVG(temp)
  FROM Measurements
```

| obsdate | temp |
|---|---|
| 20 Mar 2019 | 26 C |
| 20 Mar 2019 | 26 C |
| 21 Mar 2019 | 26 C |

```
SELECT obsdate, AVG(temp)
  FROM Measurements
 GROUP BY obsdate
```

| obsdate | temp |
|---|---|
| 20 Mar 2019 | 25 C |
| 21 Mar 2019 | 28 C |

## Queries with GROUP BY and HAVING

- In SQL, we can “partition” a relation into groups according to the value(s) of one or more attributes:

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
```

### Example: Filtering Groups with HAVING clause

- GROUP BY Example:
  - What was the average temperature at each date?

```
SELECT obsdate, AVG(temp)
FROM Measurements
GROUP BY obsdate;
```

  - What was the average temperature at each date?

```
SELECT obsdate, AVG(temp)
FROM Measurements
GROUP BY obsdate
HAVING COUNT(temp) >= 10
```

- Note: Predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

- Find the average marks of 6-credit point courses with more than 2 results

```
SELECT uos_code as unit_of_study, AVG(mark)
FROM Assessment NATURAL JOIN UnitOfStudy
WHERE credit_points = 6
GROUP BY uos_code
HAVING COUNT(*) > 2;
```

# Database Querying and Visualisation in Python

## Data Gathering

- Two approaches:
  1. Push queries into DBMS, let it work and just retrieve query result
  2. Extract large chunks or even all data, and then analyse outside DBMS

## Reprise: Accessing PostgreSQL from Python: psycopg2

```
from sqlalchemy import create_engine
import psycopg2
import psycopg2.extras
import json
import os
import pandas as pd

credentials = "Credentials.json"

def pgconnect(credential_filepath, db_schema = "public"):
  with open(credential_filepath) as f:
    db_conn_dict = json.load(f)
    host = db_conn_dict['host']
    db_user = db_conn_dict['user']
    db_pw = db_conn_dict['password']
    default_db = db_conn_dict['user']
    try:
      db = create_engine('postgresql+psycopg2://'+db_user+'
:'+db_pw+'@'+host+'/'+default_db, echo = False)
      conn = db.connect()
      print('Connected successfully.')
    except Exception as e:
      print("Unable to connect to the database.")
      print(e)
      db, conn = None, None
    return db, conn
```

## Querying PostgreSQL from Python

- How to execute an SQL statement on a given connection ‘conn’

- some utility function (query()) helps to distinguish between executing commands and queries with results which are then returned as DataFrame

```
def query(conn, sqlcmd, args=None, df=True):
  result = pd.DataFrame() if df else None
  try:
    if df:
      result = pd.read_sql_query(sqlcmd, conn,
      params=args)
    else:
      result = conn.execute(sqlcmd, args).fetchall()
      result = result[0] if len(result) == 1 else result
    except Exception as e:
      print("Error encountered: ", e, sep='\n')
    return result
```

- Example: Retrieving some data from the database

```
# connect to your database
conn = pgconnect()

# prepare SQL statement
query_stmt = "SELECT * FROM Sensors"

# execute query and show first few rows of result
query_result = query(conn, query_stmt)
query_result.head()

# prepare another SQL statement including a placeholder
query_stmt2 = "SELECT COUNT(*) FROM Measurements WHERE
              stationid = %(station)s"

# execute query and print result
param = {'station': 409204}
query_result = query(conn, query_stmt2, param)
query_result

# cleanup
conn.close()
```

## Data Visualisation from SQL in Python

```
%matplotlib inline
import matplotlib.pyplot as plt

# connect to your database
conn = pgconnect()

# prepare (multiline) SQL statement:
# 'How many sensor measurements did each site report
# today?'
query_stmt = '''SELECT sitename, COUNT(*) AS
                sensor_readings
                FROM Measurements JOIN Stations USING
                (stationid)
                WHERE obsdate = CURRENT_DATE
                GROUP BY sitename'''

# execute query
query_result = query(conn, query_stmt)

# visualise result
plt.bar(query_result['sitename'],
        query_result['sensor_readings'])
plt.title('Sensor Readings per Station')
plt.xlabel('Measurement Station')
plt.ylabel('Number of Sensor Readings')
plt.grid()

#cleanup
conn.close()
```