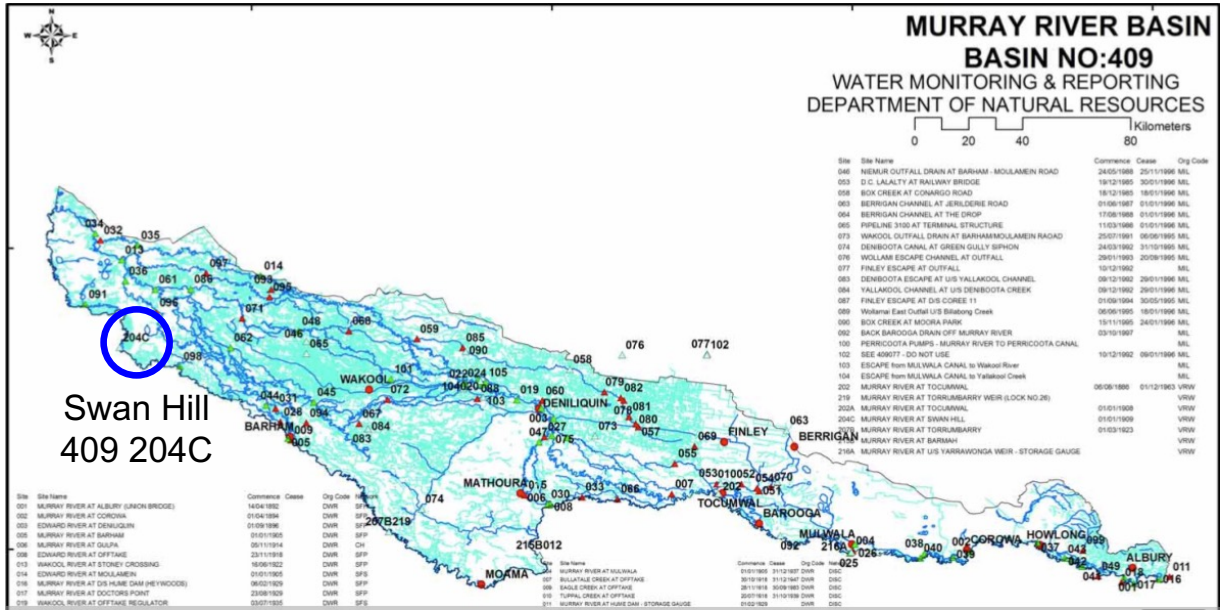
COMP5310 - Week 2

Data Transformation and Storage

Example: Water Data about Murray River Basin in NSW

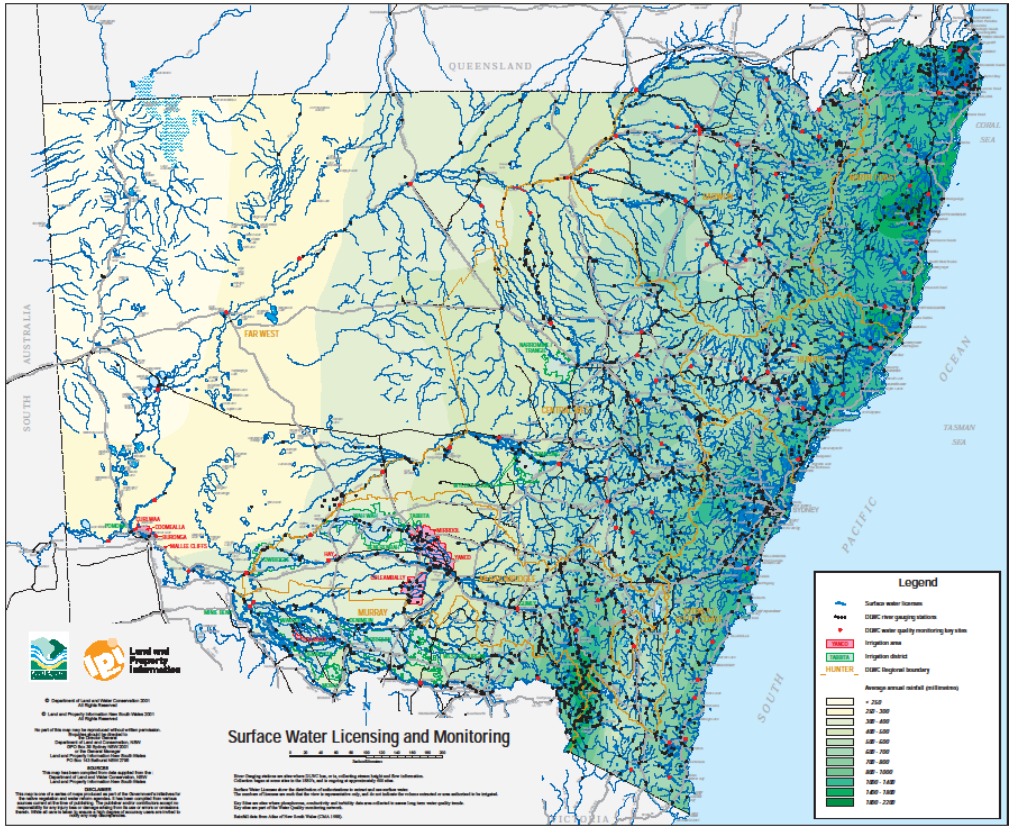


- automatic monitoring stations
- periodically send data about water level, water flow, water temperature, salinity…

[Source: mdba.gov.au]
[https://realtimedata.waternsw.com.au]

Murray River Basin Data Set

- Water measurements:
  - automatic monitoring stations
  - that are distributed over a larger area
    - (say 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



File-based Approaches

- Just put the data in files
  - eg. csv text files
  - eg. spreadsheet files

} other people could need access to these files → has changes quite often

- Anyone who wants to do some analysis, gets a copy of the files (eg. in email, or download from a web site) ✓
- Analysis might be done by writing formulas or creating charts in a spreadsheet, or by writing Python code (eg. with pandas) ✓

Example: Spreadsheet Files

Murray-waterinfo.nsw.gov.au.xls

	A	B	C	D	E	F	G	H	I
1	Station	Date	Level (m)	MeanDischarg	Discharge (ml/d)	Temp (C)	EC @ 25C (us/cm)		
272	409204C	1-Apr-09	0.713	2821.487	2773.949	21.558	54		
273	219018	1-Apr-09	-0.173	0					
274	409017	1-Apr-09	2.331	7152.066	8499.806	20.921	45		
275	409204C	2-Apr-09	0.698	2721.779	2667.749	21.833	53.5		
276	219018	2-Apr-09	-0.098	0	0				
277	409017	2-Apr-09	2.497	8972.182	10741.82	21.167	45.766		
278	409204C	3-Apr-09	0.677	2609.139	2552.696	22.194	54.458		
279	219018	3-Apr-09	0.04	0	0				
280	409017	3-Apr-09	2.638	10596.43	9263.902	21.505	47.51		
281	409204C	4-Apr-09	0.653	2470.194	2409.639	22.102	>55		
282	219018	4-Apr-09	0.166	0.198	0.371				
283	409017	4-Apr-09	2.472	8684.356	7817.866	21.569	49.823		
284	409204C	5-Apr-09	0.637	2373.525	2358.659	20.633	51.75		
285	219018	5-Apr-09							
286	409017	5-Apr-09	2.389	7734.209	7744.308	21			
287	409204C	6-Apr-09	0.637	2368.798	2390.783	20			
288	219018	6-Apr-09	--	--	0.239				
289	409017	6-Apr-09	2.403	7883.954	7861.138	20			
290	409204C	7-Apr-09	0.637	2372.584	2358.659	19			
291	219018	7-Apr-09	0.166	0.159	0.119				
292	409017	7-Apr-09	2.39	7747.374	7649.435	20			
293	409204C	8-Apr-09	0.628	2310.373	2271.601	18			
294	219018	8-Apr-09	0.162	0.105	0.091				
295	409017	8-Apr-09	2.368	7511.15	7299.264	19			
296	409204C	9-Apr-09	0.615	2221.495	2185.795	18			
297	219018	9-Apr-09	0.164	0.127	0.122				
298	409017	9-Apr-09							
299	409204C	10-Apr-09	0.601	2131.538	2101.01	18			
300	219018	10-Apr-09	0.163	0.117	0.116				
301	409017	10-Apr-09	--	--	6228.199				
302	409204C	11-Apr-09	0.591	2096.919	2086.492	19			

WaterInfo | Stations | Sensors

Ready [Source: www.waterinfo.nsw.gov.au]

Murray-waterinfo.nsw.gov.au.xls

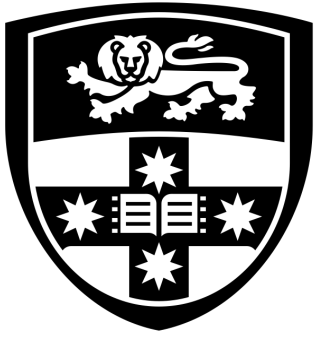
	A	B	C	D	E	F	G	H
1	Basin No	Site	Site Name	Long	Lat	Commence	Cease	Org Code
2	409	001	Murray River at Albury (Union Bridge)	146.8957 E	36.0929 S	14/04/1892		DWR
3	409	002	Murray River at Corowa	146.3954 E	36.0076 S	01/04/1894		DWR
4	409	003	Murray River at Denuquin	144.9663 E	35.5301 S	01/09/1896		DWR
5	409	005	Murray River at Barham	144.1235 E	35.6304 S	1/1/1905		DWR
6	409	204C	Murray River @ Swan Hill	143.5680 E	35.3318 S	1/1/1909		VRW
7	409	017	Murray River @ Doctors Point	146.9401 E	36.1129 S	8/23/1929		DWR
8	409	019	Wakool River at Offtake Regulator	144.8846 E	35.4985 S	7/3/1935		DWR
9	409	202	Murray River @ Tocumwal	145.5596 E	35.8151 S	06/06/1886	12/1/1963	VRW
10	219	018	Murray River @ Quaama	149.8526 E	36.4762 S	7/12/1966		DWR
11	...							
12								
13								
14	Organisation Codes							
15	Code	Organisation						
16	DNR	NSW Department of Water and Energy (and predecessors)						
17	DWR	NSW Department of Water and Energy (and predecessors)						
18	MIL	Murray Irrigation Ltd						
19	PWD	Manly Hydraulics Laboratory						
20	QWR	Qld Department of Natural Resources and Water						
21	SCA	Sydney Catchment Authority						
22	SMA	Snowy Mountains Authority						
23	SWB	Sydney Catchment Authority						
24	VRW	Vic Government						
25								

WaterInfo | Stations | Sensors

Ready Sum=0

Quality Issues with File-based Approaches

- Data quality is left to the users doing the right thing
- Eg keep meta-data somewhere, and keep it up-to-date
- Eg keep backups, manage sharing
  - Multiple copies (“redundancy”), hard to know which is most-up-to-date
- Eg prevent changes that introduce inconsistency or violate integrity properties

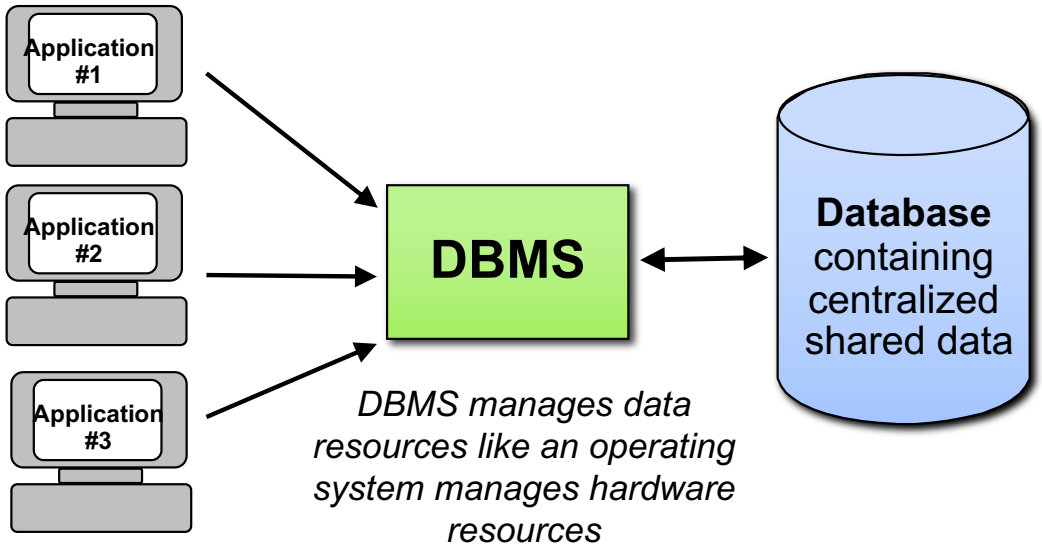


THE UNIVERSITY OF
SYDNEY

Databases

The Database Approach

- Central repository of shared data
- Data is managed by software: *Database Management System (DBMS)*
- Data is accessed by applications or users, always through the DBMS



What is a Database?

A database is a shared collection of logically related data and its description.

The database represents the **entities** (real-world things),
the **attributes** (their relevant properties), and
the logical **relationships** between the entities.

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

Advantages of Databases (continued)

- **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

Relational Databases

- **Relational data model** is the most widely used model today
  - Main concept: **relation**, basically a table with rows and columns
  - Every relation has a **schema**, which describes the columns, or fields

- This sounds like a spreadsheet, but as we will see, it has some differences

Relation Database Systems

- store data in **tables** as **rows** with multiple **attributes**
- rows *of the same format* form a 'table' (**relation: a set of tuples**)
  - Every relation has a **schema**, which describes the columns, or fields, and their types
- A relational database is a collection of such tables (which typically are related to each other by **key** attributes)
- Example:

primary key →

Student				
sid	name	email	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

schema

data

Primary Key

- A primary key is a unique attribute (or combination of attributes) which the database uses to identify a row in a table.
- It is a unique (typically auto-incrementing) ID which is NEVER NULL
  - (NULL has the special meaning in databases of “unknown” or “not given”)
- A primary ID number will only ever be issued once

Author		
authorId	**given_name**	**family_name**
1	Charles	Dickens
2	Virginia	Woolf

Foreign Key

— PK
--- FK

- When we need to refer to a record in a separate table we reference its ID as a *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.

Books		
bookID	**title**	**authorID**
1001	Orlando	2
1002	David Copperfield	1

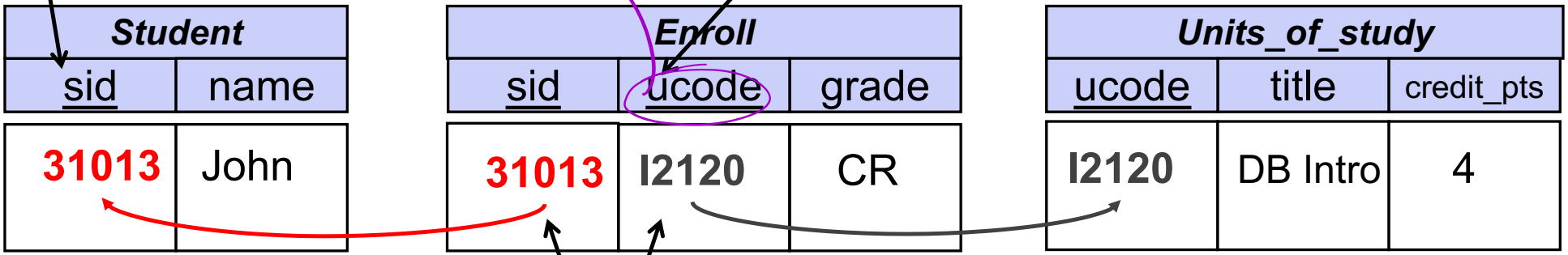
Could refer to an author table with ID as the PK.

Example: Relational Keys

Primary key identifies each tuple of a relation.

FK in Enroll table

Composite Primary Key consisting of more than one attribute.

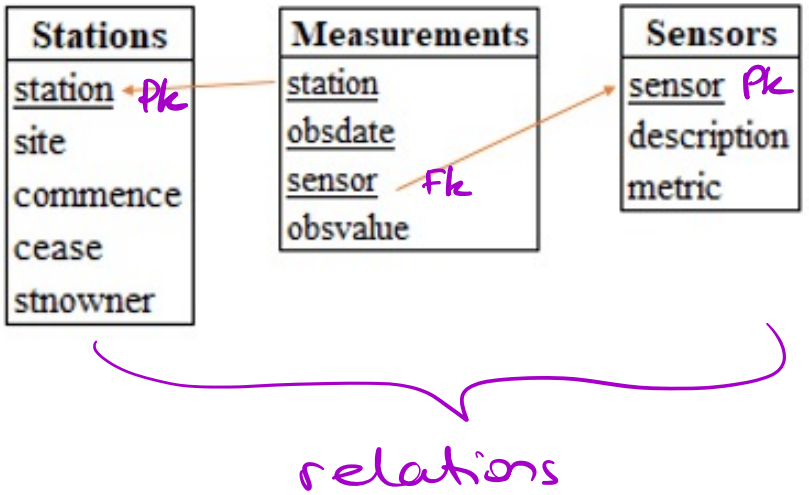


Foreign key is a (set of) attribute(s) in one relation that `refers' to a tuple in another relation (like a `logical pointer').

Schema Diagrams

WARNING: database system notations have various representations

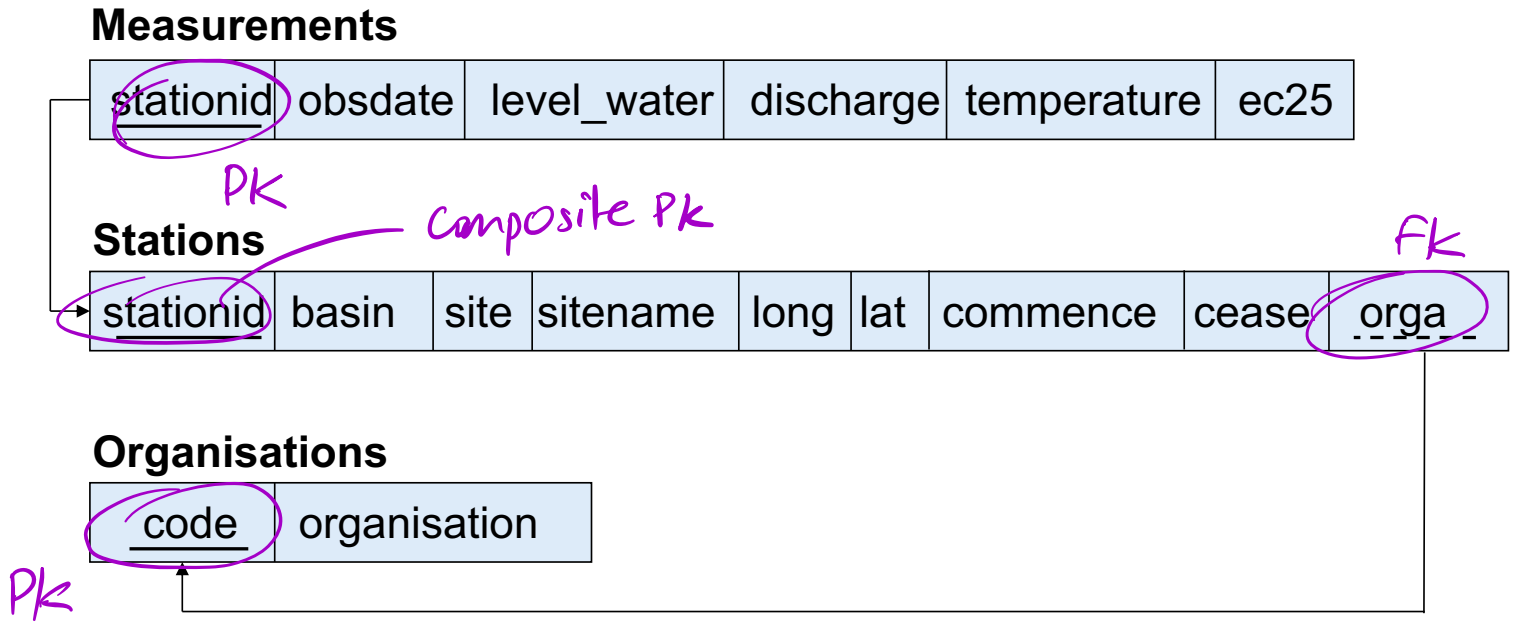
- A *normalised* relational database tries to avoid redundancies
  - Every fact ideally stored only once
  - That's some difference to spreadsheet where lots of data gets repeated and then tends to become inconsistent
- Different graphical notations used to visualise a schema
  - Also: Entity-Relationship-Diagrams



WaterInfo Example as a Relational Database – Option 1

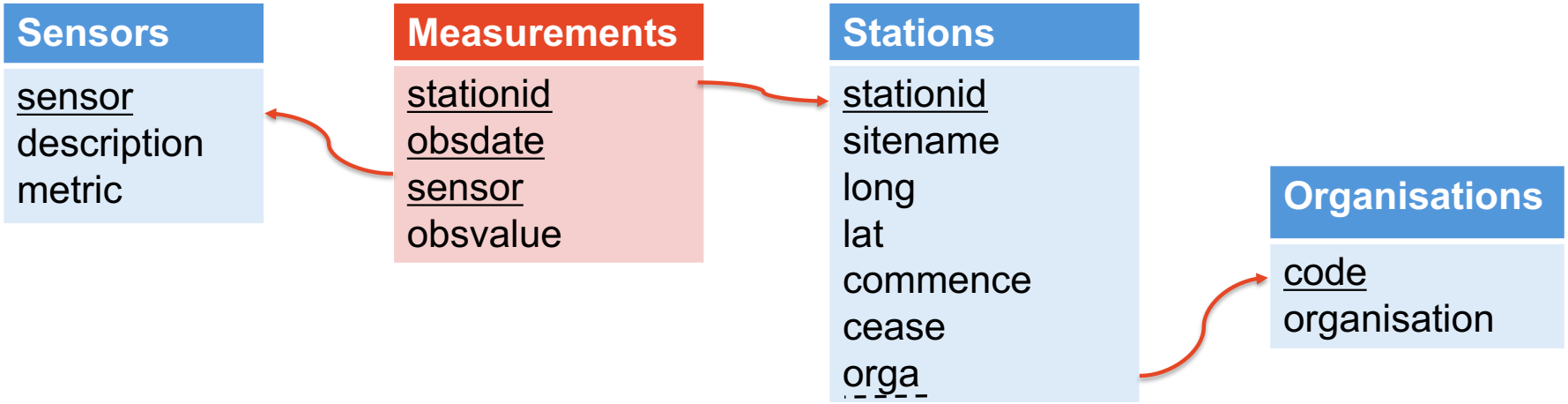
- **Design Option 1**: Straight-forward 1:1 mapping
  - Because a spreadsheet is in principle a table, we can always map it 1:1
  - Some problems though: e.g. the Station <-> Basin+Site mapping
  - A lot of NULL values for any missing measurement or if no sensor

recall Wk1 of COMP3311...

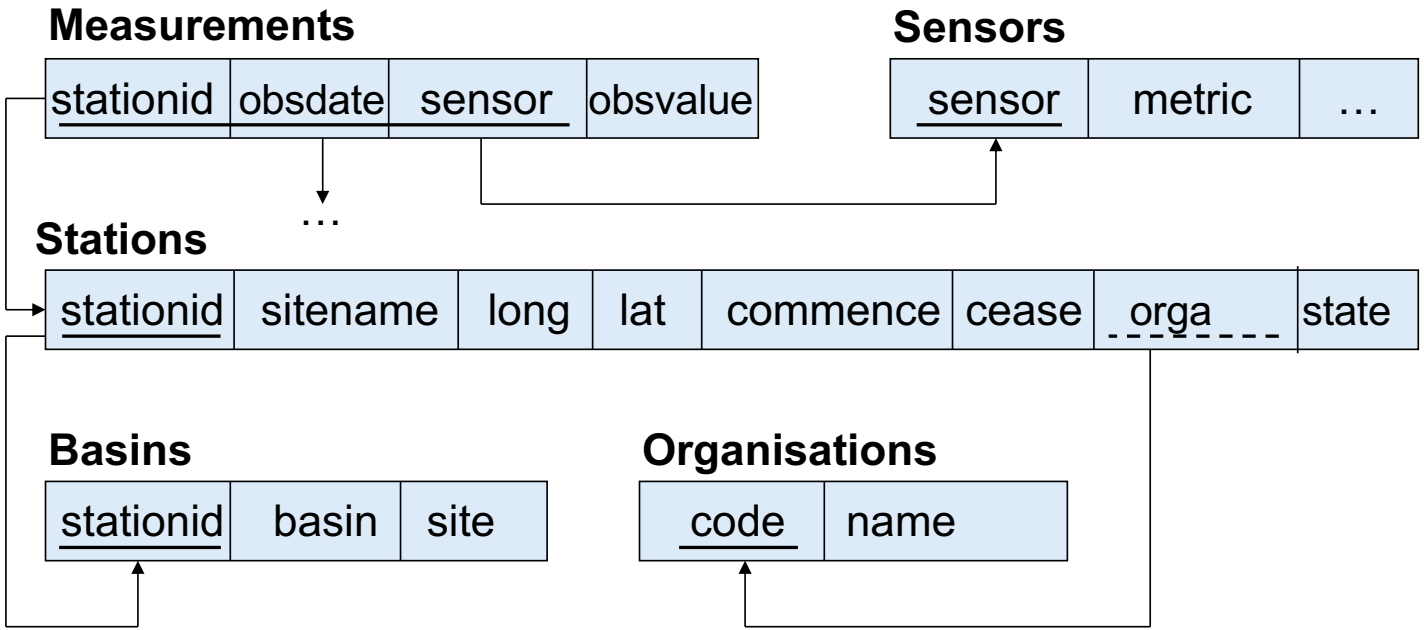


Modeling our Water Data Set – Option 2

- **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



Another Notation for this Design



Some Remarks

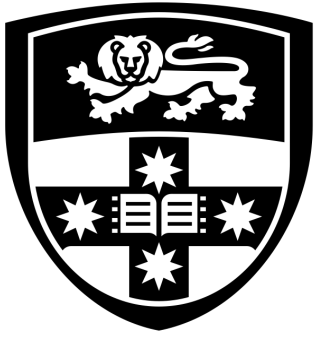
since relations are set-based.

- Not all spreadsheet 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.

    we might prefer PKs to be appearing first in the relation, but it does not matter.
  - 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)

    ↳ for database system to be normalized, we may need to have multiple tables
  - The order of the rows is immaterial



THE UNIVERSITY OF
SYDNEY

Data Storage with Python

Data Storing

- Where are we now? (on Python)
  - 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

Visualisations

Approach 1: Command Line Tools

- For example, Postgresql 's **psql** provides 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…

Query Language

Issues with DB Loader Tools

- 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? Yes!

psycopg2 for e.g.

Approach 2: Python Loading Code

- Built-in functionality in Pandas: to_sql()
  - Obtain data as DataFrame df (eg. df = pd.read_csv(*file*))
  - Execute **df.to_sql(***table, con***)** on a database connection *con*
- This creates a new table based on the header row from the Dataframe

- Can also combine with creating an own table with own integrity constraints
  - Connect to the database
  - Define a string with the appropriate CREATE TABLE command
  - Use conn.execute(string) to do this in the dbms
  - Execute **df.to_sql(** *table*, *con*, if_exists="append" **)** on db connection

- Pros: flexibility; use pandas data cleaning and transformation tools
- Cons: needs to be hand-coded

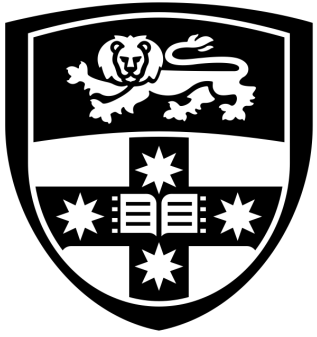
Approach 2: Python Loading Code Example

- Example: Creating a table and loading some data

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

# 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 EXISTS Organisations (
                         code         CHAR(3) PRIMARY KEY,
                         organisation VARCHAR(150)
                       )"""
conn.execute(organisation_schema)

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

utility function which we encapsulates db connection handling

Actual data storage



THE UNIVERSITY OF
SYDNEY

creating DB with SQL

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`

DDL → create, drop or alter table

DML → selecting stuff

Table Constraints and Relational Keys

- When creating a table, we can specify ***Integrity Constraints*** for columns
  - eg. domain types per attribute, or **NULL / NOT NULL** constraints

char, varchar, int etc.

- **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

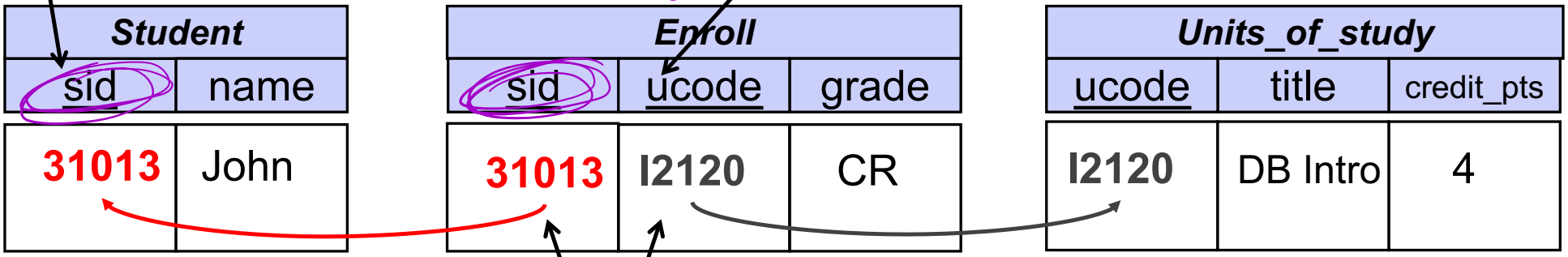
Recall ...

Primary key identifies each tuple of a relation.

Composite Primary Key consisting of more than one attribute.

ucode however is a composite PK since it is an attribute on Enroll, but PK on Units_of_study

sid here is not a composite primary key, because it is PK in both student and Enroll relation



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
- 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)
  );
```

seen this...

NULL → not default as NULL, but is allowed to be null

if not...

Data Types vs Types of Data

Data Types → domain types (atleast in databases)

Types of Data → statistical types (refer back to week 1 lecs)

- **Data Types** in SQL or Python differ from (statistical) **Types of Data**
- **Data Types** – specifies what can be stored in a field. For example, a field whose data type is a number field can store only numerical data.
  - databases actually calls this also **domain types**
  - Note that this does not state whether this numerical data is categorial, ordinal, ratio etc..
- Examples:

```
CREATE TABLE FootballPlayer (
    name        VARCHAR(100),
    team        VARCHAR(100),
    jersey_no   INTEGER,
    rating      INTEGER,
    year_born   INTEGER,
    salary      INTEGER
)
```

could be 1 digit no.

may only have max of 2 digits.

categorical

ordinal

interval

ratio

four fields with the same INT *data type*, but each represents a different *type of data*

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'
```

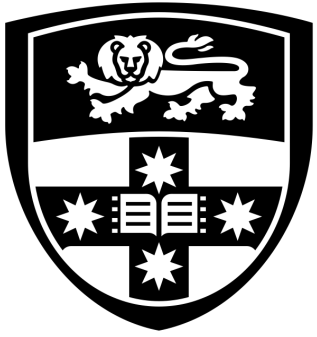
Importing Data into a DBMS – some Tips

- Add constraints and indexes **_after_** importing data
  - This also includes primary keys and foreign keys
- Bulk import multiple rows at once
  - E.g. Pandas to_sql() supports this with the options **method='multi'** and **chunksize**
- Run ANALYZE command afterwards
  - To update internal statistics

chunk together multiple rows then insert. This helps with performance

Final Thoughts

- 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
  - Needs to be installed and maintained
    (though much better nowadays for SQLite and PostgreSQL)
- What's the benefit?
  - Sharing! Large data sets!
  - Querying with SQL will give us some leverage too – and useful in other contexts too.



THE UNIVERSITY OF
SYDNEY

Go through SQL/Python exercise after watching Demo.

Querying Databases with SQL

SQL (Structured Query Language) Example

- The *working-horse* command: **SELECT – FROM – WHERE**
- retrieves data (rows) from one or more tables of a relational database that fulfill search condition
- Example 1:

```
SELECT *
  FROM Stations
```

- Example 2:

```
SELECT sitename, orga, commence
  FROM Stations
 WHERE stationId = 409001
```

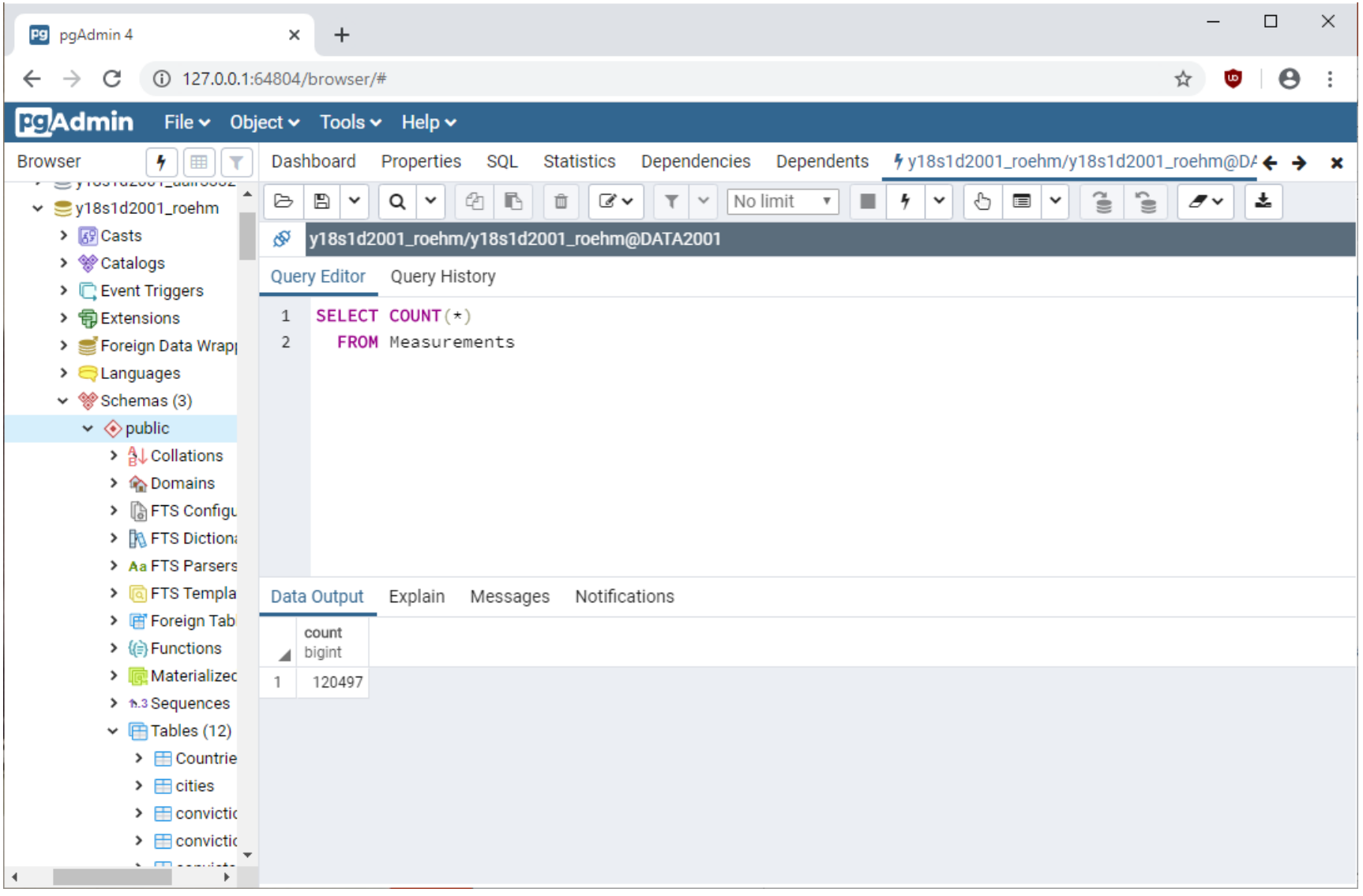
- Example 3:

```
SELECT COUNT(*)
  FROM Stations
 WHERE orga = 'SCA'
```

Note: SQL is case-insensitive and additional spaces+newlines are ignored; use this to format a query for better readability.

Declarative Queries: "What" not "How"

- It is convenient to indicate declaratively *what* information is needed, and leave it to the system to work out *how* to process through the data to extract what you need
  - Programming is hard, and choosing between different computations is hard
- Users should be offered a way to express their requests declaratively
  - A query language can be based on logic
  - Select…where…


pgAdmin 4
127.0.0.1:64804/browser/#
pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
y18s1d2001_roehm/y18s1d2001_roehm@DA
No limit
y18s1d2001_roehm
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrapp
Languages
Schemas (3)
public
Collations
Domains
FTS Configu
FTS Dictiona
FTS Parsers
FTS Templa
Foreign Tab
Functions
Materializec
Sequences
Tables (12)
Countrie
cities
convictic
convictic
y18s1d2001_roehm/y18s1d2001_roehm@DATA2001
Query Editor
Query History
1 SELECT COUNT(*)
2 FROM Measurements
Data Output
Explain
Messages
Notifications
count
bigint
1
120497

SELECT Statement

- SQL: "Lingua Franca" of the database world
- SELECT: retrieves data (rows) from one or more tables that fulfill a search condition
- Clauses of the SELECT statement:
  - **SELECT** → Lists the attributes (and expressions) that should be returned from the query
  - **FROM** → Indicate the table(s) from which data will be obtained
  - **WHERE** → Indicate the conditions to include a tuple in the result
  - **GROUP BY** → Indicate the categorization of tuples
  - **HAVING** → Indicate the conditions to include a category
  - **ORDER BY** → Sorts the result according to specified criteria

- The result of an SQL query is a relation

More SELECT Statement Options

SQL Statement	Meaning
SELECT COUNT(*) FROM *T*	count how many tuples are stored in table *T*
SELECT * FROM *T*	list the content of table *T*
SELECT * FROM *T* LIMIT *n*	only list *n* tuples from a table
SELECT * FROM *T* ORDER BY *a*	order the result by attribute *a* (in ascending order; add DESC for descending order)

Comparison Operations

SQL commands are case insensitive, but strings are case sensitive.

- 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%';
```

Murray River followed by something... or just this string

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_DATE db system's current date
  - CURRENT_TIME db system's current timestamp
- Example:

```
SELECT *
  FROM Measurements
 WHERE obsdate < CURRENT_DATE;
```

depending on when we run this command, we will get a diff result.

Date and Time in SQL (cont'd)

- Database systems support a variety of date/time related ops
  - Unfortunately not very standardized – a lot of slight differences
- Main Operations
  - **EXTRACT(** *component* **FROM** *date* **)**
    - e.g. EXTRACT(year FROM startDate)
  - **DATE** *string* (Oracle syntax: TO_DATE(*string,template*))
    - 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

- Tuples can have missing values for some attributes, denoted by **NULL**
  - Integral part of SQL to handle missing / unknown information
  - **null** signifies that a value *does not exist*, it does *not mean "0" or "blank"*!
- The predicate **IS NULL** or **IS NOT NULL** can be used to check for nulls
  - e.g. Find measurements with an unknown intensity error value.

```
SELECT *
  FROM Measurements
 WHERE discharge IS NULL
```

- Consequence: Three-valued logic
  - The result of any arithmetic expression involving null is null
    - e.g. 5 + null returns null
  - However, (most) aggregate functions simply ignore nulls

min, max etc ...

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.

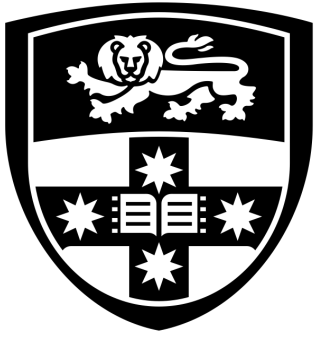
Handling NULL

- While NULL is very handy for representing incomplete or unknown data, NULLs and the resulting 3-valued 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
  - e.g. **SELECT** COUNT(*), COUNT(grade) **FROM** Students -- can produce two different counts…

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



THE UNIVERSITY OF
SYDNEY

Analysing Data from Multiple Tables

JOIN: Querying Multiple Tables

- Often data that is stored in multiple different tables must be combined
- We say that the relations are **joined**
  - **FROM** clause lists all relations involved in the query
  - join-predicates can be explicitly stated in the **WHERE** clause; do not forget it!
  - Examples:
    - Without WHERE, produces the cross-product Measurements x Stations
      (that is, every combination of rows from the two tables)

```
SELECT *
  FROM Measurements, Stations;
```

    - Find the combinations of rows with filter to make sure that particular fields match

```
SELECT *
  FROM Measurements, Stations
 WHERE Measurements.stationId = Stations.stationId;
```

Joining: Combining Data from Multiple Tables

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

```
set search_path to WaterInfo;

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

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

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>)
- Eg
  - List all details of the first three measurements including station details.

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

?? does this replace on?

  - 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;
```

Example: Join Operator

SQL offers join operators to directly formulate joins.
- makes queries a bit better readable

Example: Same query than on previous slide using JOIN operator

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

that's better :)

Natural Joins

– if we combine tuples from two tables, such as common attributes have the same type and values, returns the attributes, but not duplicates.

- The **natural join** between two tables is a join that contains any combination of tuples which match on all the *same-named attributes (*that is, *"has same values for these columns"* as implicit join condition
  - typically the case for a Primary Key – Foreign Key Relationship
  - Example:

```
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

- 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.

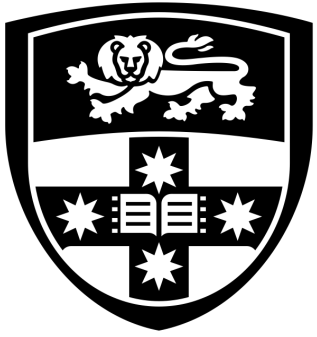
```
SELECT *
  FROM Measurements s
 WHERE sensor IN ( SELECT sensor FROM Sensors );
```

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

Tips and Tricks

- SQL provides *declarative* querying
  - can be very powerful & fast if you are familiar with SQL
  - but lacks good integration with iterative DM/ML algorithms or visualisation
    - typically still requires data processing outside dbms
- Schema required first and schema can be limiting
  - So be careful with consistency constraint and typing
  - Schema can evolve though (ALTER TABLE statement)
- Careful with NULL values as they make queries difficult
  - Better **not** to store something rather than to have NULL 'placeholder'
- RM not well fitted for *semi-structured data* such as JSON or XML
  - Yes, there are extensions nowadays (cf. XML and JSON in postgresql docu)
  - Recent rise of NoSQL databases

relational model



THE UNIVERSITY OF
SYDNEY

Summarising Data with SQL

Summarising a Database with SQL

- SQL covered so far merely allows simple exploring and retrieving of a dataset
- But we can do more with SQL:
  - Complex filtering
  - Data categorization and **aggregation**
  - **Grouping**
  - Ranking
  - Nested queries
  - Etc.

- Basis of data summarisation is the GROUP BY clause

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
…	…

do aggregate functions always go on select statement?

Example:

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

SQL Grouping

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- Example: Find the average temperature measured at each day

not per day, like we are used to.

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
```

This is different to where, which looks at rows.
Condition checked per group.

- A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.
- Note: Attributes in **select** clause outside of aggregate functions must appear in the *grouping-list*
  - Intuitively, each answer tuple corresponds to a *group,* and these attributes must have a single value per group.

Group By Overview

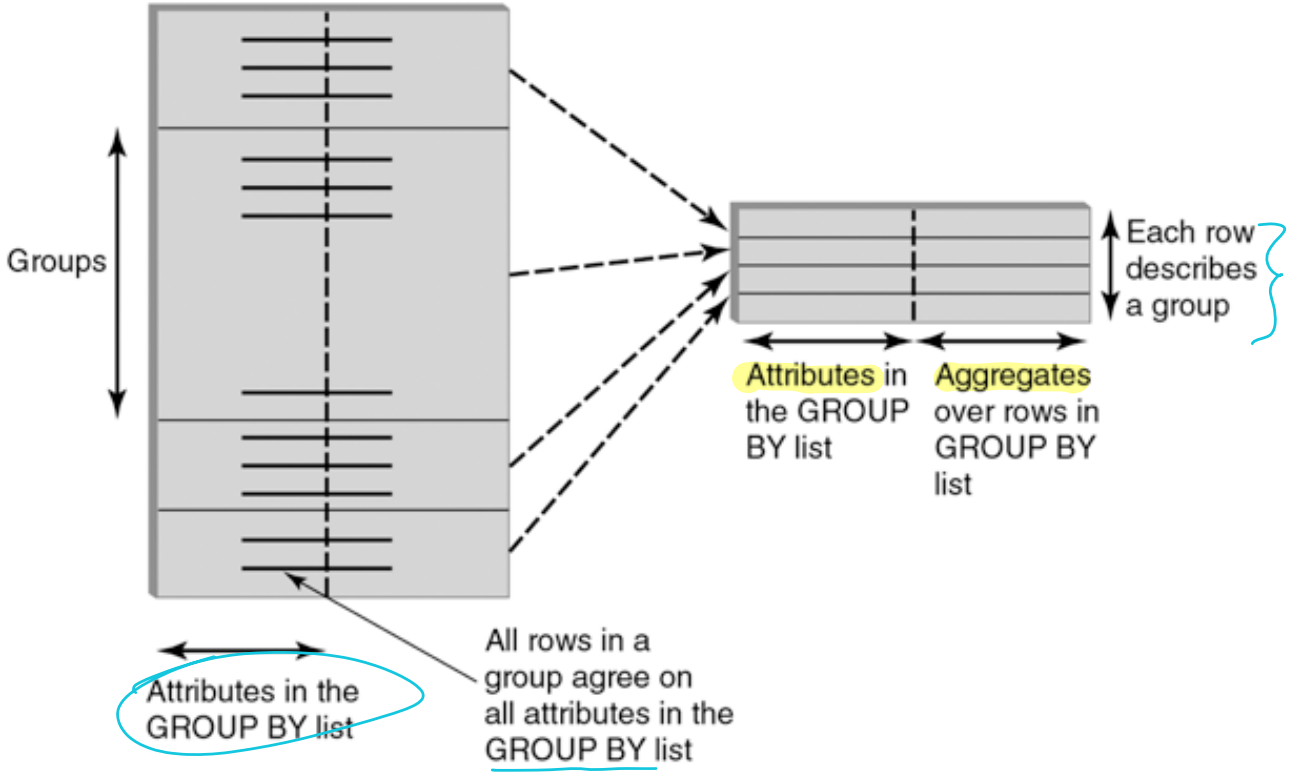


FIGURE 5.9 Effect of the GROUP BY clause.

[Kifer/Bernstein/Lewis 2006]

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
```

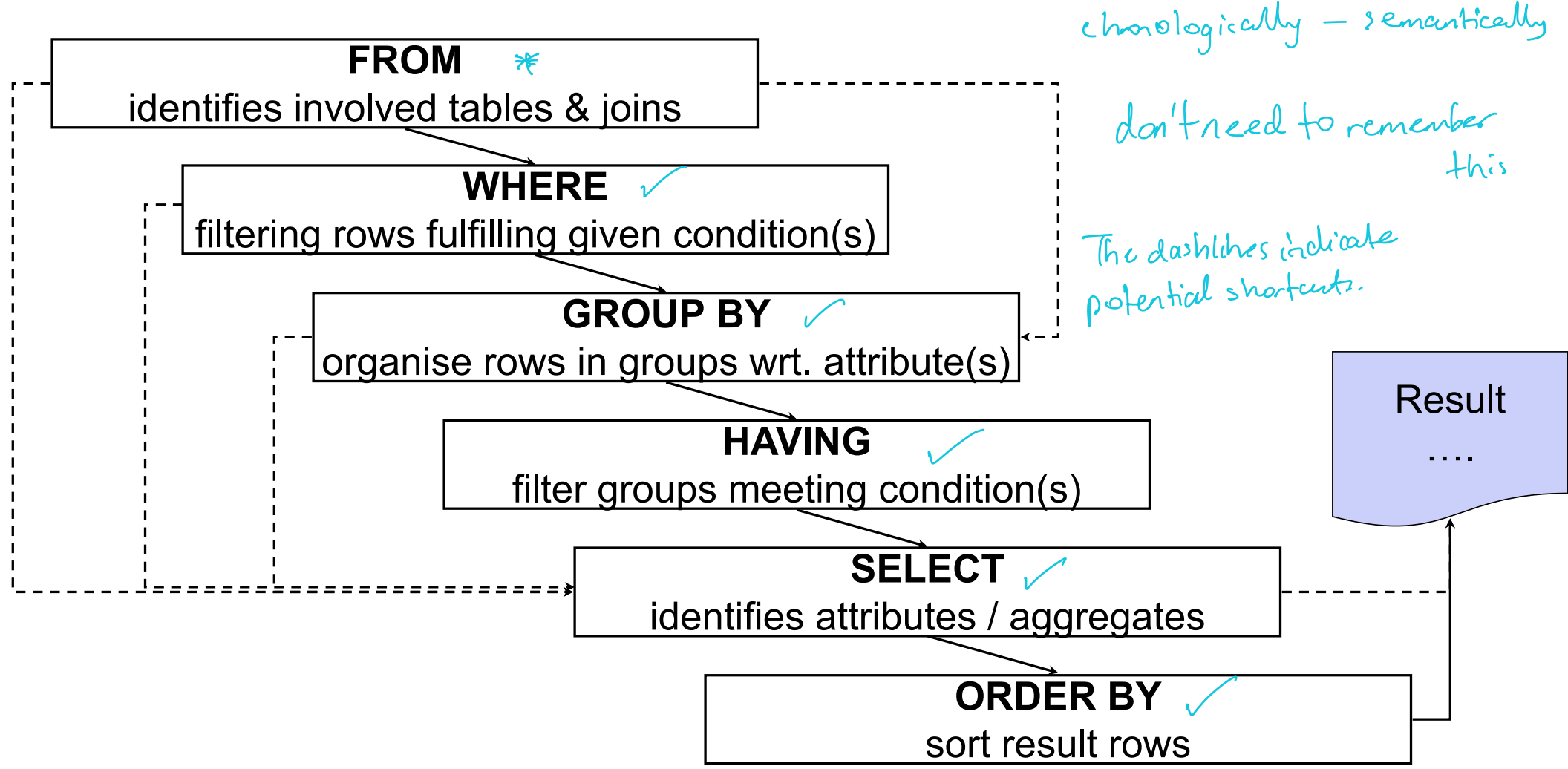
- HAVING clause: can further filter groups to fulfil a predicate
  - Example:

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

we can have where and having in an sql statement

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

Query-Clause Evaluation Order



Query Evaluation Example

- 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
```

1. Assessment and UnitOfStudy are joined

uos_code	sid	emp_id	mark	title	cpts.	lecturer
COMP5138	1001	10500	60	RDBMS	6	10500
COMP5138	1002	10500	55	RDBMS	6	10500
COMP5138	1003	10500	78	RDBMS	6	10500
COMP5138	1004	10500	93	RDBMS	6	10500
ISYS3207	1002	10500	67	IS Project	4	10500
ISYS3207	1004	10505	80	IS Project	4	10505
SOFT3000	1001	10505	56	C Prog.	6	10505
INFO2120	1005	10500	63	DBS 1	4	10500
…	…	…	….	…	…	…

2. Tuples that fail the WHERE condition are discarded

Query Evaluation Example (cont'd)

3. remaining tuples are partitioned into groups
   by the value of attributes in the grouping-list.

uos_code	sid	emp_id	mark	title	cpts.	lecturer
COMP5138	1001	10500	60	RDBMS	6	10500
COMP5138	1002	10500	55	RDBMS	6	10500
COMP5138	1003	10500	78	RDBMS	6	10500
COMP5138	1004	10500	93	RDBMS	6	10500
~~SOFT3000~~	~~1001~~	~~10505~~	~~56~~	~~C Prog.~~	~~6~~	~~10505~~
INFO5990	1001	10505	67	IT Practice	6	10505
…	…	…	….	…	…	…

4. Groups which fail the HAVING condition are discarded.

5. ONE answer tuple is generated per group

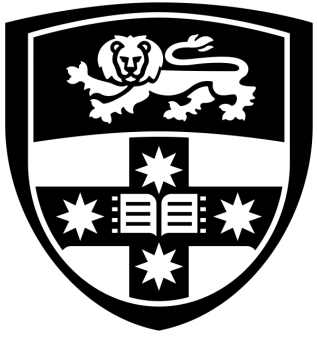
uos_code	AVG(..)
COMP5138	56
INFO5990	40.5

Question: What happens if we have NULL values in grouping attributes?

Outlook: Advanced SQL

Beyond scope of COMP5310

- OLAP Queries
  - multiple groupings by one query: GROUPING SETS, CUBE, ROLLUP
- Window Queries
  - Can define windows ('groups') per each row, incl. relative and overlapping
  - Allow to order data in a window plus new order-dependent aggregate functions
  - eg. moving average, cumulative aggregation, ranking, n-tiles, top-k
- Recursive Queries
  - Needed for working with recursive structures such as trees or graphs
- User-defined 'stored procedures'
  - UDFs (user-defined functions), UDAs (user-defined aggregates)
  - Allow to execute arbitrary code close to the data inside DBMS



THE UNIVERSITY OF
SYDNEY

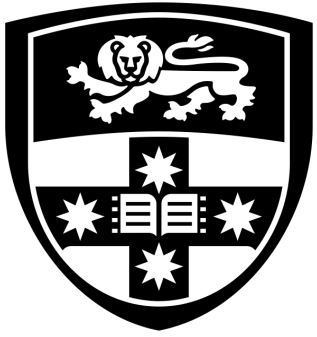
Demo: Sky Server

Use Case: SDSS SkyServer

- Website to access data from the Sloan Digital Sky Survey (SDSS)
  - astronomy survey aimed at creating a map of a large part of the universe
  - used a 2.5-meter optical telescope in New Mexico to automatically take images of about 1/3 of the night sky (northern hemisphere)
- SDSS Data Release DR16:
  - 469 million celestial objects
  - 5.79 million spectra of stars, galaxies and quasars
  - Over 125 TB of raw and processed data

Jim Gray standing next to the main SDSS telescope at Apache Point Observatory, New Mexico.

[cf. http://www.sdss.org/dr16/scope/ and https://www.sdss.org/dr16/data_access/bulk/]



THE UNIVERSITY OF
SYDNEY

Database Querying and Visualisation in Python

Data Gathering

- So far we looked at the DBMS as the *sink* for data
  (DBMS: Database Management Systems)
- But SQL databases are also a common source for data analysis
- 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

COMP3311

Reprise: Accessing PostgreSQL from Python: psycopg2

- First, we need to import psycopg2, then connect to Postgresql
  - For this, you also need some login credentials for your database
- As this is all quite system specific, best factored into a separate function pgconnect()

```
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
```

apparently missing in canvas

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
```

Querying PostgreSQL from Python (cont'd)

- 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()
```

This is where your SQL statements go

Example *range query*: query all rows of a table

parameter binding

Example *point query*: query a specific row

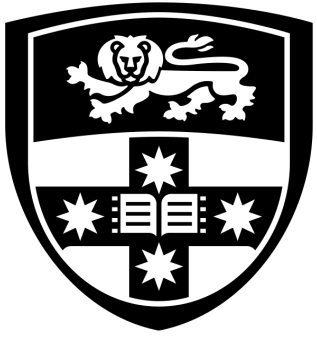
Data Visualisation from SQL in Python

```
%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()
```

Let the database do all group-by query work

Visualise result as bar chart

Standard matplotlib stuff



THE UNIVERSITY OF
SYDNEY