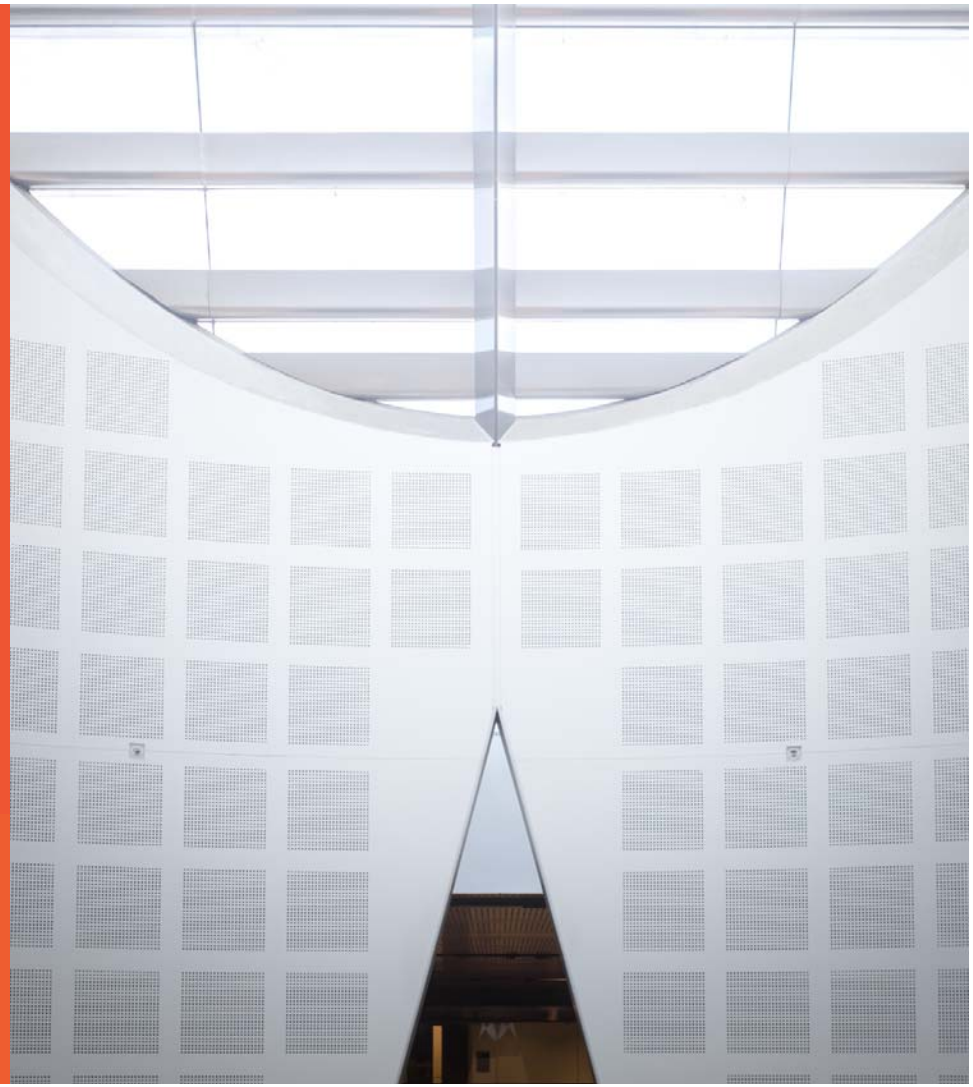
DATA2001: Data Science: Big Data and Data Diversity

W3: Accessing data in relational databases; introduction to SQL

Presented by
Dr. Matloob Khushi
School of IT

THE UNIVERSITY OF SYDNEY



Overview of Week 2

THE UNIVERSITY OF SYDNEY

Last Week: Data analysis with Python

Last Week's Objective

Learn Python tools for exploring a new data set programmatically.

Lecture

- Data types, cleaning, preprocessing
- Descriptive statistics, e.g., median, quartiles, IQR, outliers
- Descriptive visualisation, e.g., boxplots, confidence intervals

Readings

- Data Science from Scratch: Ch 4-5

Exercises

- matplotlib: Visualisation
- numpy/scipy: Descriptive stats

TODO in W2/W3

- Explore the survey data

Today: Accessing data in relational databases and introduction to SQL

Objective

To be able to extract a data set from a database, as well as to leverage on the SQL capabilities for in-database data summarisation and analysis.

Lecture

- Data Gathering reprise
- SQL querying
- Summarising data with SQL
- Statistic functions support in SQL

Readings

- Data Science from Scratch, Ch 23

Exercises

- Creating database / tables
- SQL Querying
- Data Summarization using SQL

TODO in W3/4

- Data Analysis using SQL

Databases

THE UNIVERSITY OF SYDNEY

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.

Why Databases?

- Size of data
- Ease of updating data
- Accuracy
- Security
- Redundancy

Databases in the Internet Age…

- Ebay (in 2005)
  - More than 100 back-end databases
  - ca. 5 billion SQL/day
- Salesforce.com
  - Ca. 1.3 billion transactions per day
- Pinterest
  - Started with 1 MySQL database
  - Now 180+ web servers, 240 API servers, 88 MySQL DBMSs, …
- Wikipedia: (as of Feb 2015 - http://stats.wikimedia.org/EN/Sitemap.htm)
  - over 400 servers
  - 237 languages, millions of articles
    (English: 4.8 million articles with more than 12 GB data)
  - 3 million edits/month
- 2010: World of Warcraft uses 1.3 petabytes of storage
- 2012: Facebook's Hadoop cluster has >100 Petabyte storage

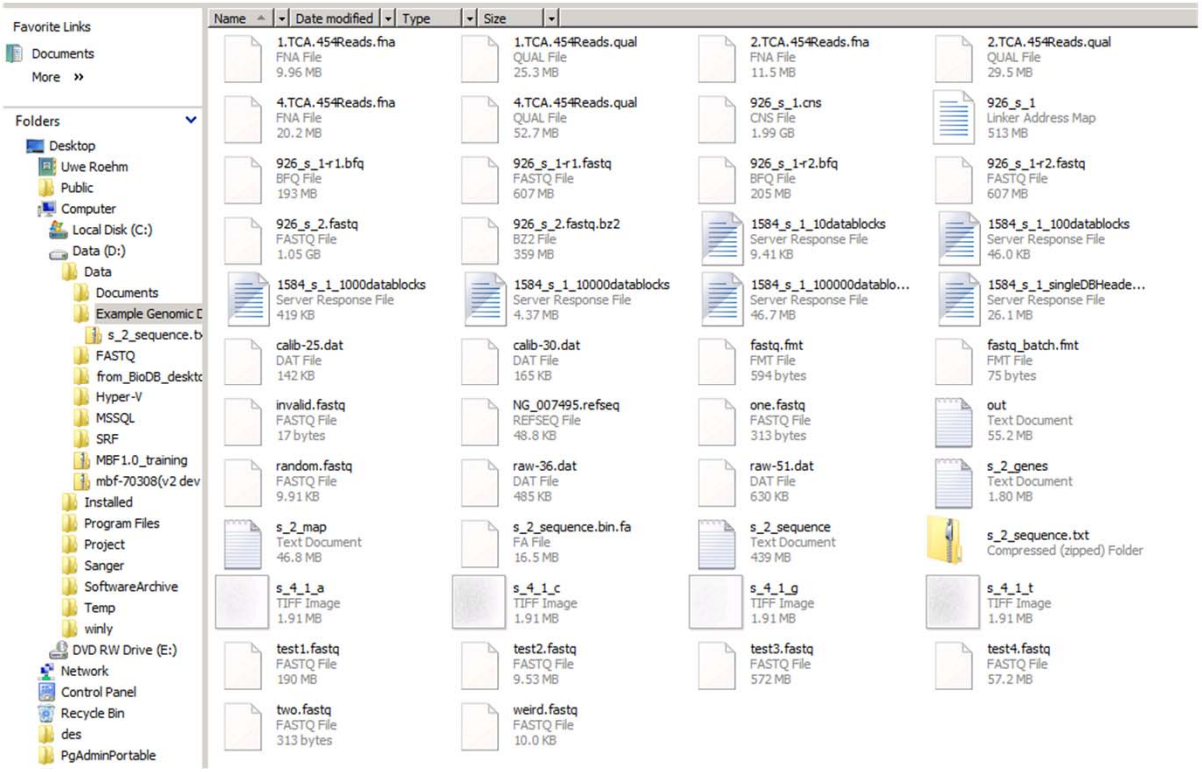
Data-Intensive Scientific Discovery

- Scientific Research as it evolves over time:
  - **Experimental** (thousands of years ago)
  - **Theoretical** (few hundred years ago)
  - **Computational** (last few decades)
  - **Data-Intensive** (termed by the late *Jim Gray*)

    [Tony Hey, et al (ed.): *The Fourth Paradigm: Data-Intensive Scientific Discovery*, Microsoft Research, 2009.]

- eScience: "IT meets scientists"
  - Modern scientific instruments allow to automatically collect Petabytes of scientific results and data that is shared world-wide
    - e.g. CERN's Large Hydron Collider (LHC): 200 PB data as of 2012
  - At the essence this means: **Data-intensive research**
    To base theories and results purely on the analysis of this data.
- **eScience depends on effective data management**

File-Processing still an Issue Today



Example: Perl-Script in Bioinformatics

```
#!perl
use strict;
my $in  = shift or die; # illumina sequence file
my $out = shift or die; # output file for binned sequences
open (FH, "$in") or die;
my @lines = <FH>;
close FH;
my %reads;
for my $line (@lines) {
    chomp $line;
    my $read;
    if ($line =~ /\:([A|T|C|G]+)\:/ || $line =~ /^([A|T|G|C]+)/) {
        $read = $1;
        if ($reads{$read}->{count}) {
            $reads{$read}->{count}++;
        } else
       {  $reads{$read}->{count} = 1;    }
 }}
@lines = "";
open (OUT, ">$out") or warn;
my $m =1; # for rank
for my $read (sort { $reads{$b}->{count} <=> $reads{$a}->{count} } keys
    %reads){
    print OUT ">$reads{$read}->{count}\-$m\n$read\n";  # count, rank
    $m++;
}
close OUT;
```

A relatively simple Perl script from a Bioinformatics project that processes a file (binning of unique ‘short-reads’)

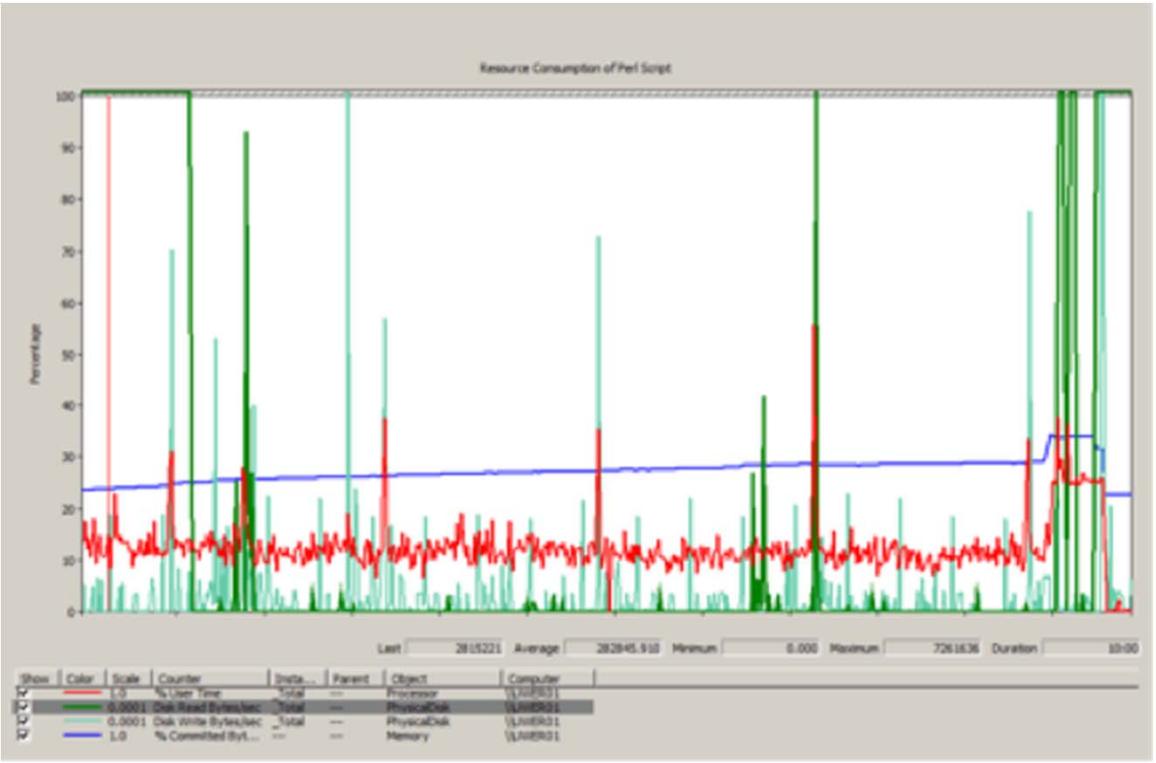
Example: The Same in SQL

```
INSERT INTO UniqueRead
  SELECT ROW_NUMBER()OVER(ORDER BY COUNT(*)DESC),
         COUNT(*),
         r_sequence
    FROM ShortReads
   WHERE r_e_id=1 AND r_sg_id=2 AND r_s_id=1
         AND CHARINDEX('N',r_sequence) = 0
   GROUP BY r_sequence
```

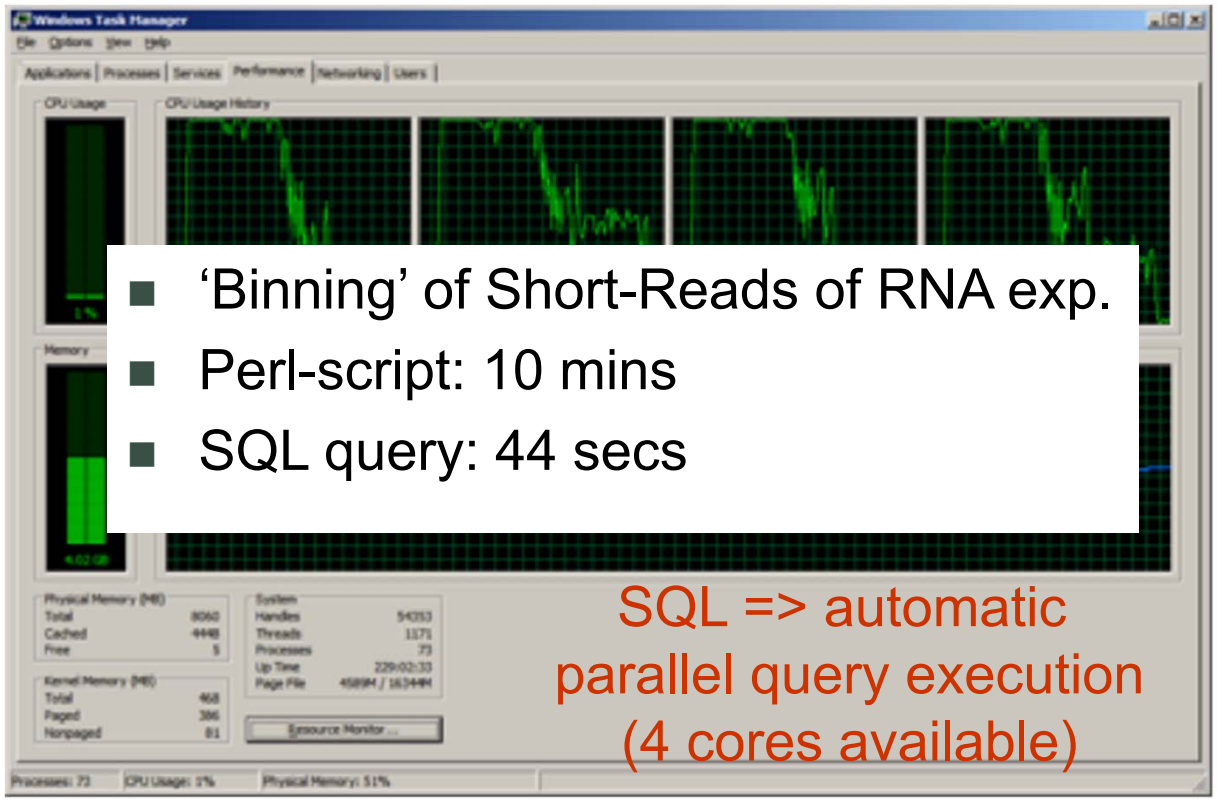
- Basically a group-by/aggregation query
- Tricky part is to determine the 'rank' of a result

Perl Script Performance

Note that the CPU load is only 25% - Why?



SQL Query Performance



- ‘Binning’ of Short-Reads of RNA exp.
- Perl-script: 10 mins
- SQL query: 44 secs

SQL => automatic parallel query execution (4 cores available)

Disadvantages of File Processing

- Program-Data Dependence
  - All programs contain full descriptions of each data file they use
- Data Redundancy (Duplication of data)
  - Different systems/programs have separate copies of the same data
  - No centralized control of data => **Integrity problems!**
- Limited Data Sharing
  - Required data stored in several, (potentially incompatible) files.
- Lengthy Development Times
  - Programmers must design their own file formats
  - For each new data access task, a new program is required.
- Excessive Program Maintenance
  - 80% of information systems budget

Problems with Data Dependency

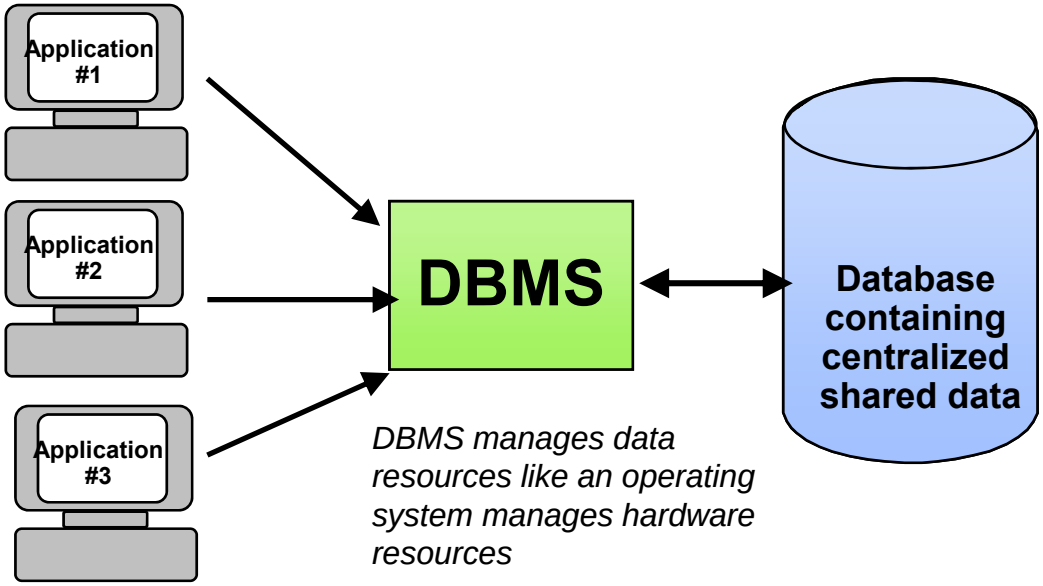
- Each application programmer must **maintain their own data**
- Each application program needs to include code for the **metadata of each file**
- Each application program must have its **own processing routines** for reading, inserting, updating and deleting data
- **Lack of coordination** and central control
- **Non-standard** file formats

Problems with Data Redundancy

- Waste of space to have duplicate data
- Causes more maintenance headaches
- The biggest Problem:
  - When data changes in one file, could cause inconsistencies
  - Compromises **data integrity**

Solution: The Database Approach

- Central repository of shared data
- Data is managed by a DBMS
- Stored in a standardized, convenient form



DBMS manages data resources like an operating system manages hardware resources

Main Advantages of Databases

- **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

- **Minimal Data Redundancy**
  - Leads to increased data integrity/consistency

Advantages of Databases (cont'd)

- 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
- Improved Data Quality
  - Integrity constraints, data validation rules
- Better Data Accessibility/ Responsiveness
  - Use of standard data query language (SQL)
- Security, Backup/Recovery, Concurrency
  - Disaster recovery is easier

Key database concepts

- **Table –** an arrangement of related information stored in columns and rows.
- **Field / attribute –** coloumn in a table, contains homogenous set of information.
- **Field data types** - kind of data that can be stored in a field. For example, a field whose data type is Text can store data consisting of either text or number characters, but a Number field can store only numerical data.
- **Primary Key (PK) –** a field in a table whose value is uniquely identifies each record in the table. A PK cannot be null.
- **Record –** A row in table.

Primary Key

- A primary key is a unique attribute which the database uses to identify a row in a table.
- It is a unique, auto-incrementing ID which is filled in by the database - in other words it is NEVER NULL.
- A primary ID number will only ever be issued once

Id	Name	Surname
1	Charles	Dickens
2	Virginia	Woolf

Foreign Key

- 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.

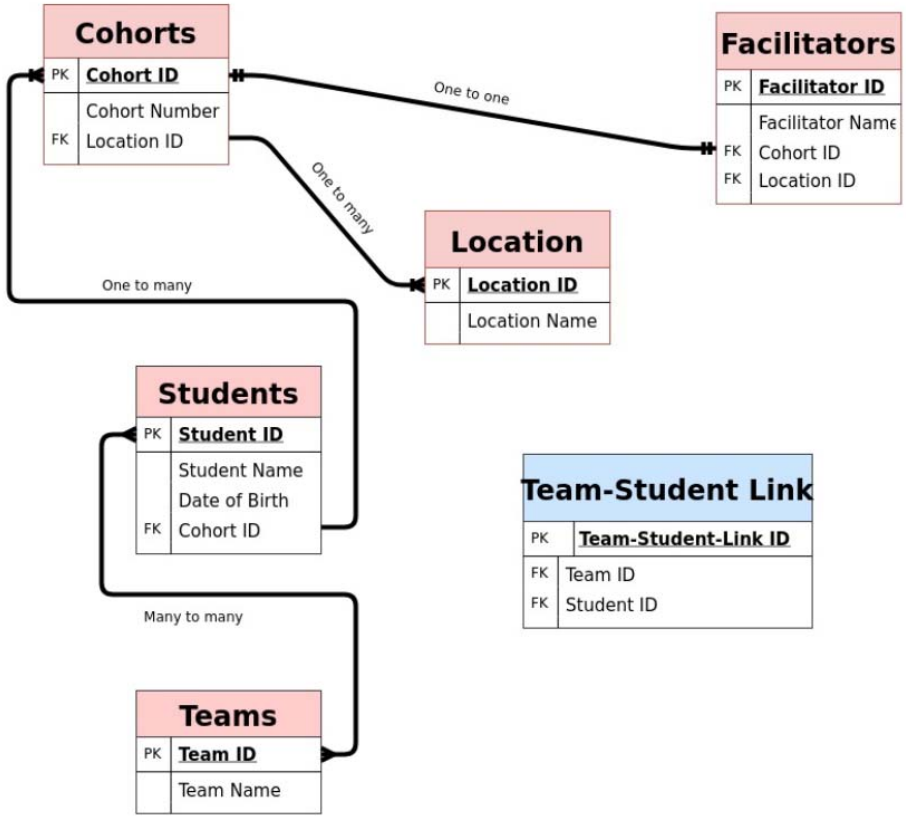
Id	Book	Author ID
1	Orlando	2
2	David Copperfield	1

Entity Relationship

- **One-One Relationship (1-1 Relationship):** One-to-One (1-1) relationship is defined as the relationship between two tables where both the tables should be associated with each other based on only one matching row.
- **One-Many Relationship (1-M Relationship):** The One-to-Many relationship is defined as a relationship between two tables where a row from one table can have multiple matching rows in another table.

Entity Relationship Diagram

- 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
- Example: "Star Schema"



Relation Database Management System (RDBMS)

- stores data 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
- A relational database is a collection of such tables
  (which typically are related to each other by key attributes)
- Example:

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

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 a search condition
- Example 1:

```
SELECT *
  FROM Student
```

- Example 2:

```
SELECT name, email
  FROM Student
WHERE sid=5312666
```

- Example 3:

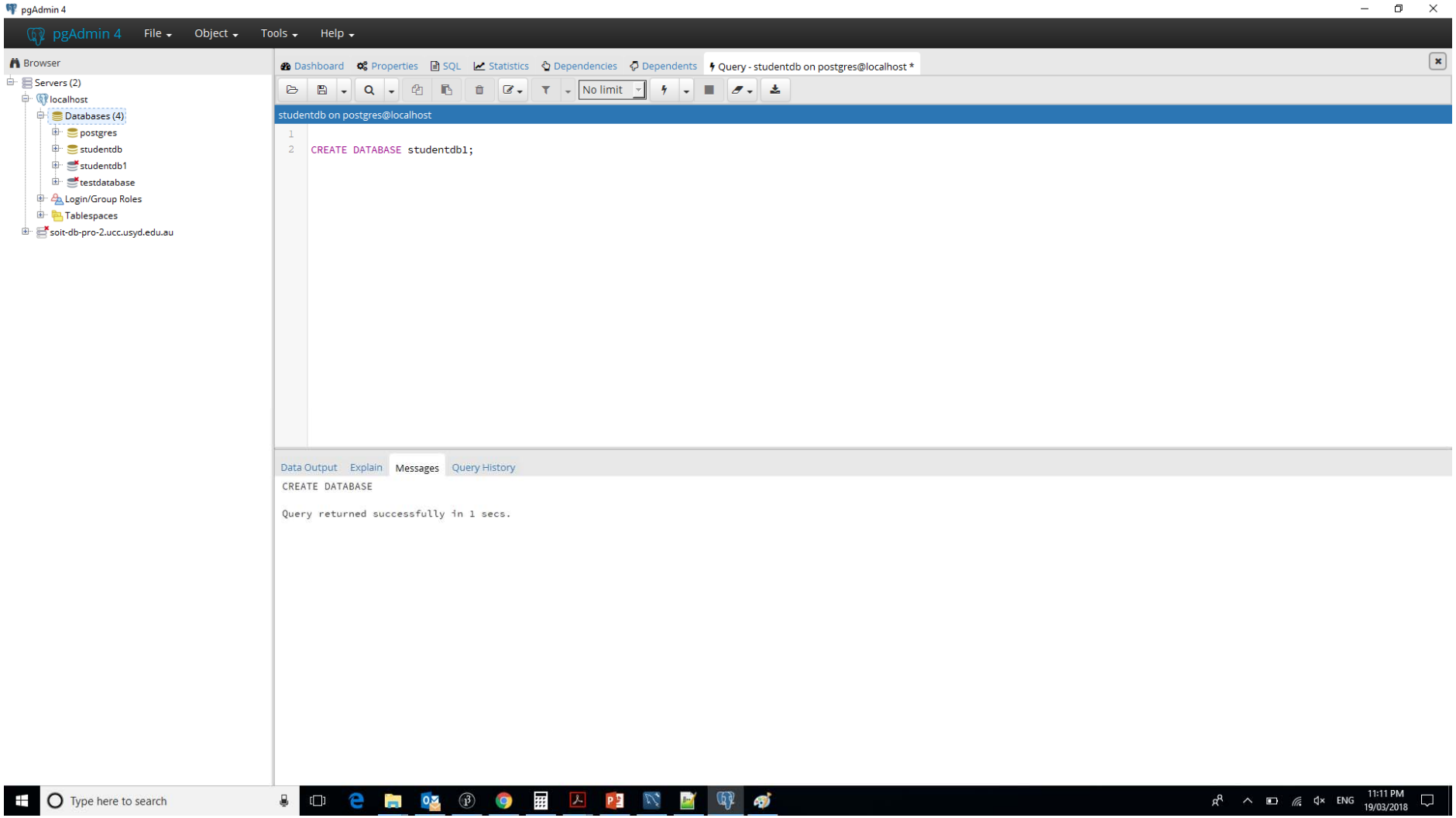
```
SELECT COUNT(*)
  FROM Student
 WHERE gender='f'
```

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


pgAdmin 4
File
Object
Tools
Help
Browser
Servers (2)
localhost
Databases (4)
postgres
studentdb
studentdb1
testdatabase
Login/Group Roles
Tablespaces
soit-db-pro-2.ucc.usyd.edu.au
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Query - studentdb on postgres@localhost *
No limit
studentdb on postgres@localhost
1
2
CREATE DATABASE studentdb1;
Data Output
Explain
Messages
Query History
CREATE DATABASE
Query returned successfully in 1 secs.
Type here to search
ENG
11:11 PM
19/03/2018

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)

SQL Data Types

- Integers
  - Standard integer arithmetic and comparisons available
- Floats, Numeric
  - Floating point numbers with many mathematical operators and functions
- Strings (CHAR, VARCHAR)
  - SQL string literals must be enclosed in single quotes ('like this')
  - CHAR: fixed length; VARCHAR: variable length strings up-to max length
  - String comparison is case-sensitive
  - Pattern matching with LIKE operator and % and _ placeholders
  - String concatenation: || (eg. 'hello ' || 'there')
- Date, Timestamp

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 Table
 WHERE textTypeField LIKE 'H%';
```

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 Epoch
 WHERE startDate < CURRENT_DATE;
```

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*
    - Oracle syntax: TO_DATE('01-03-2012', 'DD-Mon-YYYY')
  - **+/- 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 FieldName 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

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	false	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

- Result of **where** clause predicate is treated as false if it evaluates to unknown
  - e.g: **select** sid **from** enrolled **where** grade = 'unknown'
    ignores all students without a grade so far

SQL Aggregate Functions

SQL Aggregate Function	Meaning
COUNT(*attr*) ; COUNT(*)	Number of *Not-null-attr ; or* of <u>all</u> 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
…	…

Summarising Data with SQL

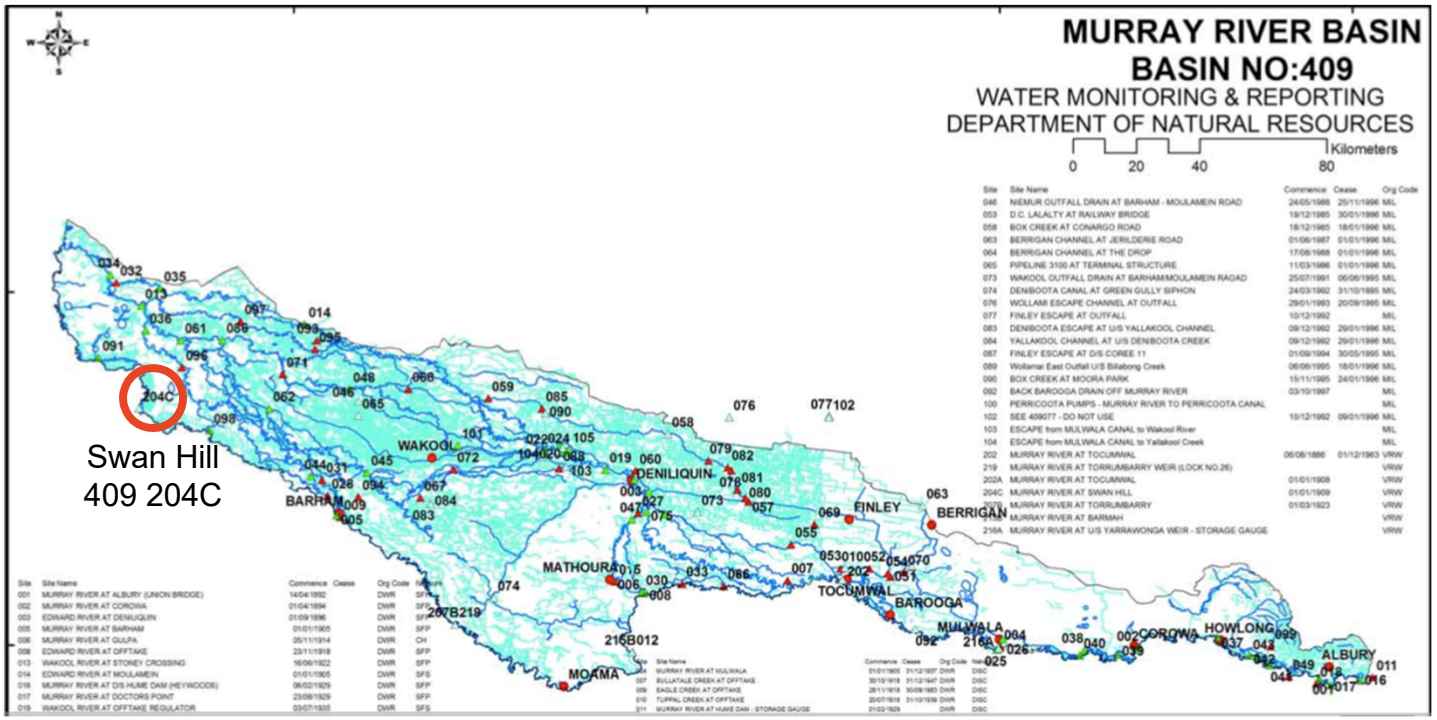
THE UNIVERSITY OF SYDNEY

Summarising a Database with SQL

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

- Basis of data summarisation is the the GROUP BY clause

Example: Murray River Basin in NSW



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

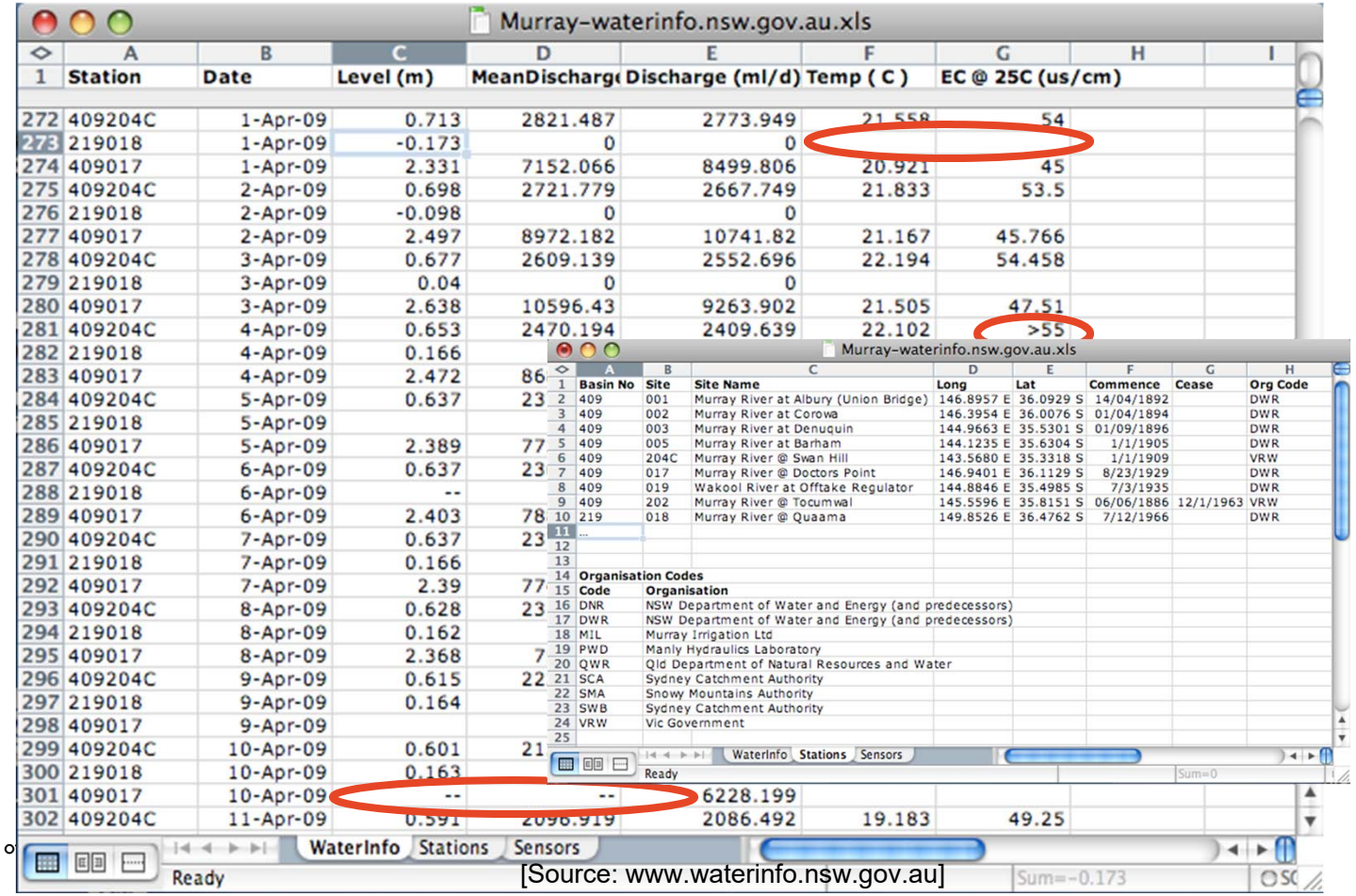
Approach 1: CSV Files

- Just dump the data in tab-delimited text files
- Send those around

- Pros:
  - On Unix systems, you can apply all kinds of command-line tools
  - Easy to import into a spreadsheet program

- Cons:
  - No clear standard for CSV files, especially with regard to integration of meta-data (CSV files are not self-describing)
  - No security, no data integrity, easy to manipulate or to corrupt

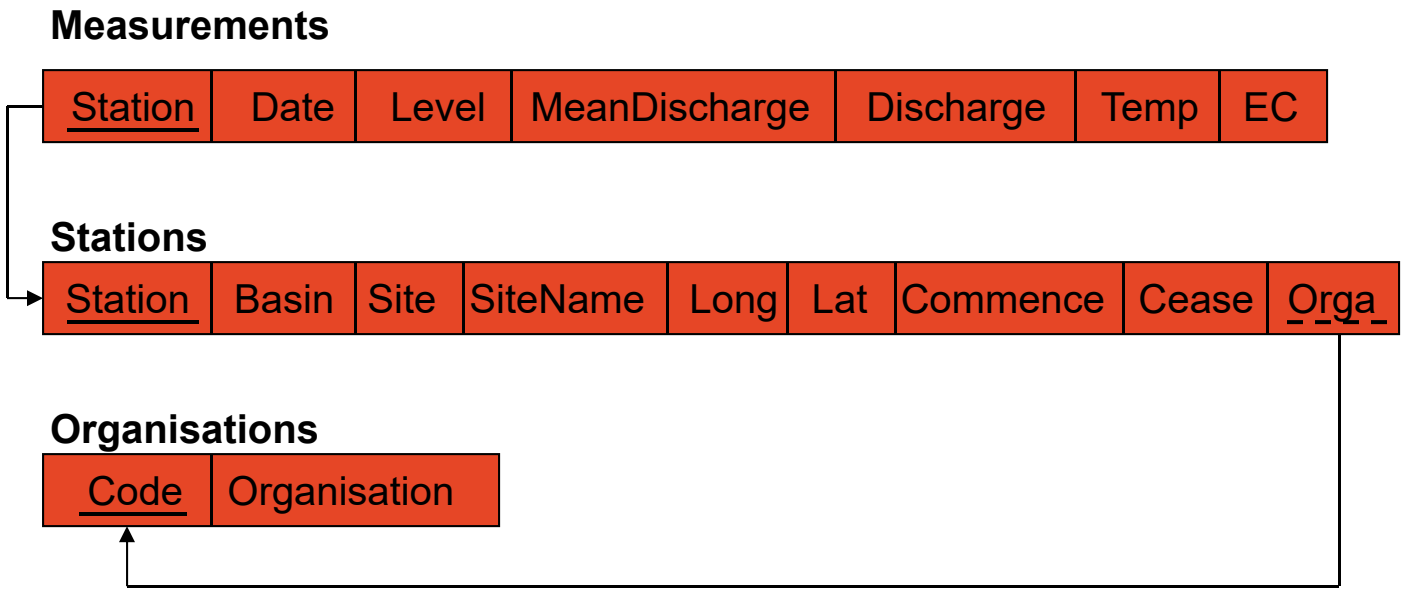
Approach 2: Spreadsheet



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

Approach 3: Relational Database

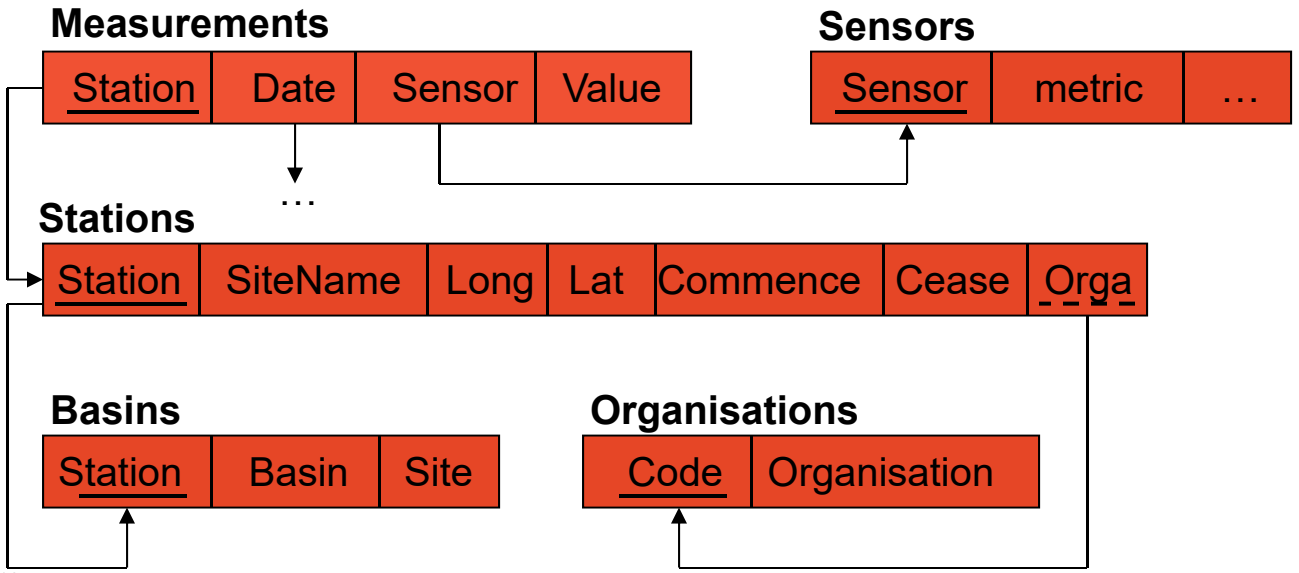
- Option 1: Straight-forward 1:1 mapping
  - As a spreadsheet is in principle a table, we can always map it 1:1
  - Some problems though: e.g. the Station <-> Basin+Site mapping



- A lot of NULL values for any missing measurement or if no sensor

Approach 3: Relational Database (contd)

- Option 2: OLAP-like Schema Schema ('Snowflake Schema')
  - measurements are the facts, rest describes the dimensions
  - measurement lines get 'folded' into separate rows of the fact table
  - allows us to avoid NULLs as much as possible, but hard to read



JOIN: Querying Multiple Tables

- Often data that is stored in multiple different relations 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:
  - Produces the cross-product *Table1* x *Table2*

```
SELECT *
  FROM Table1, Table2;
```

  - Find the start date and end date of all epochs abbreviated with 'nov04':

```
SELECT Field1, Field2
  FROM Table1 t1, Table2 t2
 WHERE t1.field1Id = t2.field2Id
       AND t1.field3Id = t2.field4Id;
```

SQL Join Operators

- SQL offers join operators to directly formulate the natural join, equi-join, and the theta join RA operations.
  - R **natural join** S
  - R [**inner**] **join** S **on** <join condition>
  - R [**inner**] **join** S **using** (<list of attributes>)
- These additional operations are typically used in the from clause
  - List all details of the first three measurements including galaxy data.

```
SELECT *
  FROM Galaxy JOIN Measurement USING (gid)
 LIMIT 3;
```

  - Which measurements were taken at stationed 409204?

```
SELECT *
  FROM measurements m INNER JOIN stations s ON m.stationid=s.id
 WHERE stationid = 409204;
```

Background: More Join Operators

- Available join types:
  - **inner join**
  - **left outer join**
  - **right outer join**
  - **full outer join**
- Join Conditions:
  - **natural**
  - **on** <*join condition*>
  - **using** <*attribute list*>

e.g: *Student* **inner join** *Enrolled* **using** *(sid)*

inner join result						
sid	name	birthdate	country	sid2	uos_code	grade
112	*'A'*	*01.01.84*	*India*	*112*	*SOFT1*	*P*
200	*'B'*	*31.5.79*	*China*	*200*	*COMP2*	*C*

e.g : *Student* **left outer join** *Enrolled* **using** *(sid)*

left outer join *result*						
sid	name	birthdate	country	sid2	uos_code	grade
112	*'A'*	*01.01.84*	*India*	*112*	*SOFT1*	*P*
200	*'B'*	*31.5.79*	*China*	*200*	*COMP2*	*C*
210	*'C'*	*29.02.82*	*Australia*	*null*	*null*	*null*

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 company and total amount of sales

Sales Table

company	amount
IBM	5500
DELL	4500
IBM	6500

```
SELECT Company, SUM(Amount)
  FROM Sales
```

company	amount
IBM	16500
DELL	16500
IBM	16500

```
SELECT Company, SUM(Amount)
  FROM Sales
 GROUP BY Company
```

company	amount
IBM	12000
DELL	4500

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
```

- 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

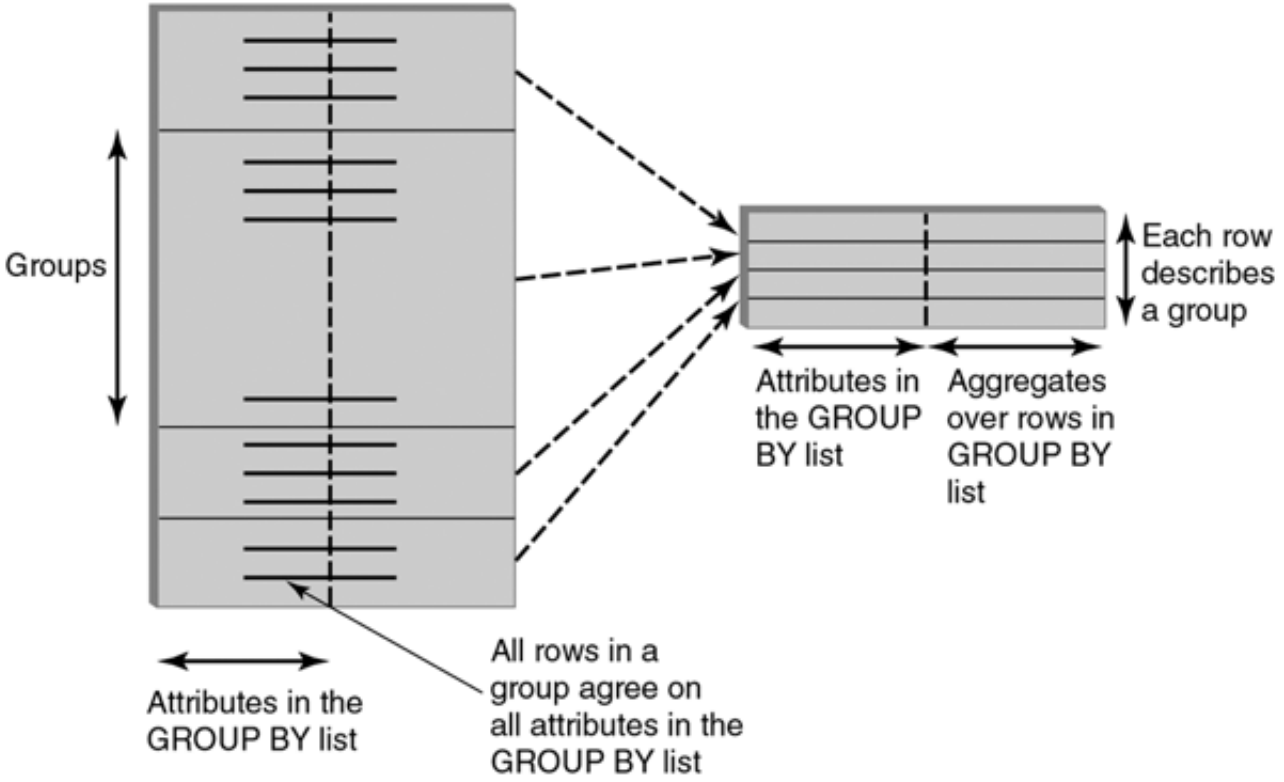


FIGURE 5.9 Effect of the GROUP BY clause.
[Kifer/Bernstein/Lewis 2006]

Example: Filtering Groups with HAVING Clause

- GROUP BY Example:
  - What was the average mark of each course?

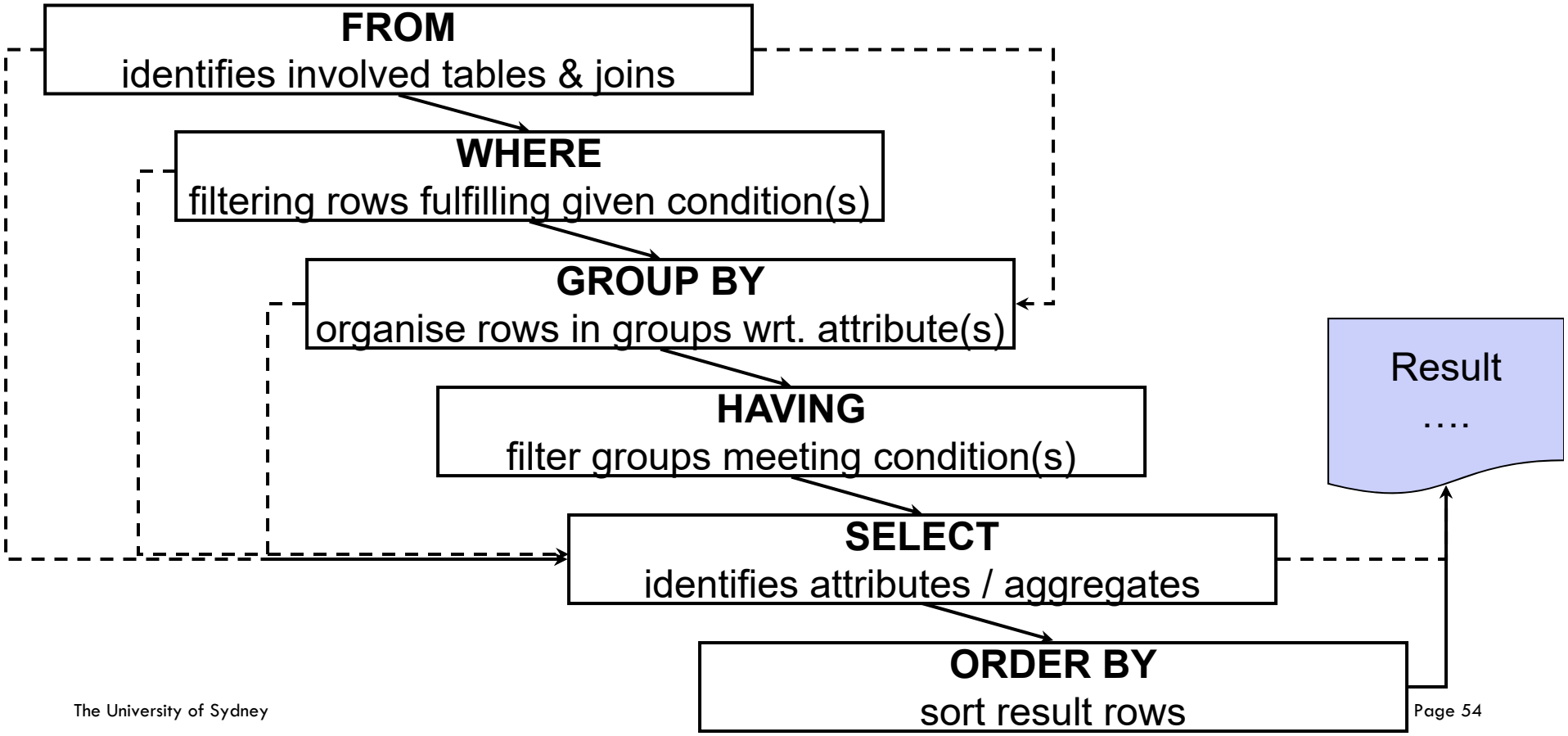
```
SELECT uos_code as unit_of_study, AVG(mark)
  FROM Assessment
GROUP BY uos_code
```

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

```
SELECT uos_code as unit_of_study, AVG(mark)
  FROM Assessment
GROUP BY uos_code
  HAVING AVG(mark) > 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

Query-Clause Evaluation Order



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

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

- 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 fcts.
  - 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

Background Info: GROUPING SETs

- SQL standard includes a generic GROUP BY extension that generalises from both *cube* and *rollup*: **GROUPING SETS**
- Allows to explicitly specify a set of grouping operations which should be performed by a single query
  - Pro: Less overhead than individual OLAP queries; single scan only
  - Con: Only supported by commercial DBMS and PostgreSQL 9.5 so far (Oracle, DB2, Microsoft SQL Server, Teradata, PostgreSQL 9.5)
- Example:

```
SELECT prod_id, cust_id, channel_id, SUM(quantity_sold)
  FROM Sales
 WHERE cust_id < 3
 GROUP BY GROUPING SETS (
    (prod_id), (cust_id, channel_id)
 );
```

Background Info: Windowing vs. Grouping

- GROUP BY
  - partitions data into a **SET** of rows
  - which all share the same partition key(s)
  - Aggregation for whole partition results into **one** aggregate per group
    - 1 group, 1 aggregation
  - set-based aggregates

- OVER … *WINDOW*
  - partitions data to a **LIST** of rows
  - from a range 'around' the current reference row
  - Aggregation over window **per each** reference row
    - 1 window, *N* aggregations
  - set-based and new list-based aggregates

Window Query Example

Rank() vs. Dense_Rank() vs. NTile() vs. Row_Number() vs. …

```
SELECT firstName, lastName, salesamt
   ,ROW_NUMBER() OVER (ORDER BY salesamt) AS RowNum
   ,RANK()       OVER (ORDER BY salesamt) AS Rank
   ,DENSE_RANK() OVER (ORDER BY salesamt) AS DenseRank
   ,NTILE(4)     OVER (ORDER BY salesamt) AS Quartile
   ,SUM(salesamt)OVER (ORDER BY salesamt) AS AcumSum       -- accumulative sum example
   ,AVG(salesamt)OVER (ORDER BY salesamt ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS MovingAvg
FROM Customer;
```

window specification example

firstName	lastname	salesamt	RowNum	Rank	DenseRank	Quartile	AcumSum	MovingAvg
Jilian	Carson	2006	1	1	1	1	6018	2006.0
Peter	Pan	2006	2	1	1	1	6018	2006.0
Shu	Ito	2006	3	1	1	2	6018	2006.0
David	McDonald	2015	4	4	2	2	10048	2009.0
Lynn	Tsofliakes	2015	5	4	2	3	10048	2012.0
Mark	Reiter	2020	6	6	3	3	14088	2016.67
Rachel	Valdez	2020	7	6	3	4	14088	2018.33
Rajit	Pak	2037	8	8	4	4	16125	2025.67

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

Review

THE UNIVERSITY OF SYDNEY

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 postgres docu)
  - Recent rise of NoSQL databases