

# CITS1402

## Relational Database Management Systems

### Video 01 — Introduction

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Data

---

- ▶ A *datum* is a single fact.

*Jack Johnson has student number 20723081  
June 2019 rainfall in Perth was 212.0mm*

- ▶ *Data* is the plural of datum, i.e., multiple facts.

*Vast amounts* of data are created and collected daily:

- ▷ People (Facebook, Apple, Amazon, Microsoft, Google)
- ▷ Science (Astronomy, Meteorology, Nuclear Physics)
- ▷ Commerce (Industry, Organizations, Purchases)
- ▷ Things (Maps, Traffic, Internet-of-Things)

Estimated world data storage as at 2022 is about 60 *zettabytes*<sup>1</sup>

---

<sup>1</sup>mega-, giga-, tera-, peta-, exa-, zetta-, yotta-.

# Drowning in data

---

Just *collecting* data is not enough:

*We are drowning in data and starved for information*<sup>2</sup>

Data needs to be *organised* in such a way that it can be used to create *information* and *insight*.

---

<sup>2</sup><https://ericbrown.com/drowning-in-data-starved-for-information.htm>

# Data Science

---

The is the main goal of *Data Science*, which involves:

- ▶ Data Collection and Storage
- ▶ Mathematics and Statistics
- ▶ Programming and Visualisation
- ▶ Data Mining and Analysis
- ▶ Machine Learning

# Databases

---

A *database* is a *structured collection* of data:

For example, a database might contain data about:

- ▶ Students, Courses, Units and Grades
- ▶ Customers, Products, Orders and Deliveries
- ▶ Doctors, Patients, Prescriptions and Drugs
- ▶ Students, Books, Periodicals and Loans

# A database management system

---

**CRUD**

A *database management system* (DBMS) is any system that enables the four basic functions:

- ▶ Create data
- ▶ Read data
- ▶ Update data
- ▶ Delete data

Relational DBMS

Collectively these four abilities are known as CRUD, so a database management system is *any system* that has CRUD.

# An old-fashioned DBMS

---

Wooden cabinets full of *index cards*, each relating to a single book.

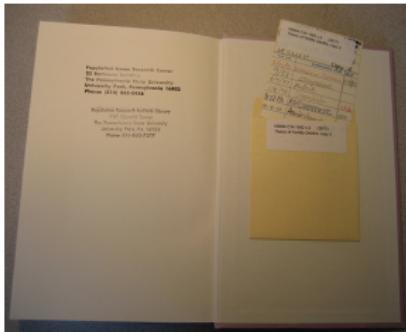


Cards are *alphabetically sorted* to allow readers to quickly find available books.

## On checkout

---

The index card was removed, the borrower's name recorded, and the due-date stamped onto a sheet of paper glued in the book.



Checked-out cards stored in *due-date order*, so easy to find when a book is returned, and quickly identify which books are overdue.

## Features of this DBMS

---

Despite being manual, the library system nevertheless displays some of the features of a relational database:

It had separate listings (catalogues) for *books* and *borrowers* with the check-out cards forming a third list *connecting* specific books to specific borrowers.

Each of the lists is *sorted* in order to permit rapid searches — in modern terminology, we would say that the lists are *indexed*.

# Types of DBMS

---

As its name suggests, CITS1402 is about *relational* database management systems.

More precisely, we are studying the database paradigm where:

*Data are stored according to the relational model, and the database is queried, updated and maintained using Structured Query Language (SQL).*

The *relational model* means that data is stored in a *collection of tables*, with the data for a single entity usually stored across multiple tables.

# Why Relational Databases?

---

The relational model of data has been the *dominant paradigm* for database management since the advent of computing.

According to db-engines.com here are the top 10 databases by popularity (as of July 2022).

Rank			DBMS	Database Model
Jul 2022	Jun 2022	Jul 2021		
1.	1.	1.	Oracle 	Relational, Multi-model 
2.	2.	2.	MySQL 	Relational, Multi-model 
3.	3.	3.	Microsoft SQL Server 	Relational, Multi-model 
4.	4.	4.	PostgreSQL 	Relational, Multi-model 
5.	5.	5.	MongoDB 	Document, Multi-model 
6.	6.	6.	Redis 	Key-value, Multi-model 
7.	7.	7.	IBM Db2	Relational, Multi-model 
8.	8.	8.	Elasticsearch	Search engine, Multi-model 
9.	9.	↑ 11.	Microsoft Access	Relational
10.	10.	↓ 9.	SQLite 	Relational

## In this unit

---

In this unit we will study:

- ▶ The *theory* underlying relational databases, and
- ▶ The *practice* of using an RDBMS with SQLite.

CITS1402  
Relational Database Management Systems  
Video 02 — Unit Details

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Unit Activities

---

- ▶ Lectures
  - ▷ All lectures pre-recorded (due to COVID-19)
  - ▷ Lectures released at start of each week
- ▶ Computer Labs
  - ▷ One 2-hour *lab period* per student
  - ▷ Choose either *online* or *face-to-face*
  - ▷ Will use SQLite, preferably on your own laptop
  - ▷ Submit some lab work on four occasions

*We do not know how many will choose online vs face-to-face, so the balance of labs and number of lab facilitators may change as semester progresses.*

# Unit Resources I

---

- ▶ help1402
  - ▷ Web-based discussion board
  - ▷ <https://secure.csse.uwa.edu.au/run/help1402>
  - ▷ Incognito posting permitted<sup>1</sup>
  - ▷ Students strongly encouraged to *answer* questions
  - ▷ *No Question Is Too Simple To Ask*
- ▶ Individual consultation
  - ▶ 10am-11am Tuesdays (after Seminar) in my office (Room 213 Maths)
  - ▶ Online option available, please wear mask in my office
  - ▶ Special Considerations all go via the Student Office

---

<sup>1</sup>but posts can be traced if absolutely necessary

## Unit Resources II

---

- ▶ Numerous textbooks cover relational databases.
- ▶ Most are *hefty tomes* covering a vast range of topics.  
*Mathematical* foundations, *CS* programming, practices and applications, *Engineering* implementation and tuning.
- ▶ All of them are slightly different to the others.
- ▶ Vast number of *tutorial websites*, blog posts etc.
  - *SQLite*

# Four books



BOOK **Practical SQL: A Beginner's Guide to Storytelling with Data** ✓

DeBarros, Anthony, San Francisco, CA, No Starch Press, Incorporated, 2018

Add tags to item

Complete Available online >



BOOK **SQL practical guide for developers** ✓

Donahoo, Michael J., Speegle, Gregory D. (Gregory David), Boston, Elsevier, c2005

Add tags to item

Complete Available online >



BOOK **Database management systems** / ✓

Ramakrishnan, Raghu., Gehrke, Johannes., 3rd ed., Boston :, McGraw-Hill, c2003.

Complete Available at Barry J Marshall Library High demand collection : 005.74 2003 DAT

EdX



BOOK **Database systems : the complete book** / ✓

Garcia-Molina, Hector., Ullman, Jeffrey D., Widom, Jennifer, Second edition.,  
Upper Saddle River, N.J. ; Pearson Education Limited, c2009., Total Pages xxvi, 1203 p. :

Complete Available at Barry J Marshall Library High demand collection : 005.74 2014 DAT

Available online >

# SQLite

---

For *hands-on learning*, we'll use:

- ▶ SQLite (version 3)      ↗ 3.35
- ▶ <https://sqlite.com/index.html> ←

This is very easy to install:



# Assessment

---

## UNIT OUTLINE

Assessment items are:

- ▶ Regular lab work (15%)  
A total of 15 SQL queries to be submitted (in four separate submissions)
- ▶ Midterm Test (15%) [In person if possible]  
Includes multiple choice part and SQL query part
- ▶ Small Project (15%)  
A short project implementing the database design and SQL techniques we have covered
- ▶ Final Examination (55%)  
Multiple choice + SQL query writing + DB theory

- SATURDAY  
- (ALTERNATE SITTING)

# Learning Advice I

---

Challenges we will *face*, but *overcome*!

- ▶ Pandemic uncertainty → LECTURES ONLINE
- ▶ Dry subject matter
- ▶ Extensive technical vocabulary
- ▶ Sophisticated concepts
- ▶ Inflexible language (SQL) — declarative
- ▶ Cryptic error messages —

## Learning Advice II

---

To overcome these challenges, focus on these simple tips.

- ▶ Know yourself
- ▶ Keep up
- ▶ Seek help
- ▶ Be realistic
- ▶ Take responsibility
- ▶ Act not react



## Learning Advice III

---

*"How to Be Successful in School: 40 Practical Tips for Students"<sup>2</sup>* by Daniel Wong is an interesting article which contains more tips about learning.

Read the article and notice how many of the tips are about being **organized** and **systematic** in your approach.

can learn organization

---

<sup>2</sup><https://www.daniel-wong.com/2018/01/30/be-successful-in-school/>

CITS1402  
RELATIONAL DATABASE MANAGEMENT  
SYSTEMS

VIDEO 03 — RELATIONAL MODEL I

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## A SIMPLE MODEL

---

Small and/or personal databases often use

- ▶ Plain Files + Text Editor, or
- ▶ Microsoft Excel.

You may even read that

*“Excel is the world’s most popular database”*

which—by some measures—has at least a grain of truth.

# AN EXCEL DATABASE

---

The 10 highest-rated movies on IMDB<sup>1</sup> (July 2022).

title_id	title	genres	premiered	rating
tt0111161	The Shawshank Redemption	Drama	1994	9.3
tt0068646	The Godfather	Crime,Drama	1972	9.2
tt0050083	12 Angry Men	Crime,Drama	1957	9
tt0071562	The Godfather: Part II	Crime,Drama	1974	9
tt0468569	The Dark Knight	Action,Crime,Drama	2008	9
tt0108052	Schindler's List	Biography,Drama,History	1993	8.9
tt0110912	Pulp Fiction	Crime,Drama	1994	8.9
tt0167260	The Lord of the Rings: The Return of the King	Action,Adventure,Drama	2003	8.9
tt0060196	The Good, the Bad and the Ugly	Western	1966	8.8
tt0109830	Forrest Gump	Drama,Romance	1994	8.8

This intuitive *tabular format* underlies the relational model.

---

<sup>1</sup>Internet Movie Database <https://imdb.com>

# RELATIONAL TERMINOLOGY

---

1970s

The *relational model* is based on *formal mathematical concepts*, so uses mathematical terminology in a precise way, e.g.

- ▶ Relation, instance, tuple, attribute, field.

In practice, *actual databases* are somewhat *loosely based* on the formal model, so more informal terminology is used, e.g.

- ▶ Table, row, column, header.

However, it is important to recognise both the formal terms and the informal terms.

# ENTITIES

---

- ▶ An *entity set*

A collection of similar objects – for example, a collection of *movies* or a collection of *books*, or a collection of *students* etc.

- ▶ An *instance* or *entity*

An actual individual object from an entity set – for example, “*The Godfather*” and “*The Shawshank Redemption*” are instances from the entity set *Movie*

a particular example

## TABLES / RELATIONS

---

A table/relation stores the data for a particular *entity set* — for example, we saw a table **Movie** that is used to store data about popular movies.

A university database may have a table called **Student** to store data about students.

A table has:

- ▶ A *name* allowing the designer and user to refer to it
- ▶ A *header row* giving names to the *columns*
- ▶ Zero or more *rows*, each row representing one entity

# Book

---

This is a tiny example of a table called Book.

Author	Title	Date
Tolstoy	Anna Karenina	1873
Steinbeck	Cannery Row	1945
Wharton	Ethan Frome	1911
Conrad	Lord Jim	1900
Kerouac	Big Sur	1962

## THE ROWS I

---

In **Movie**, each row represents *an individual movie*, while in **Book**, each row represents *an individual book*.

Author	Title	Date
Tolstoy	Anna Karenina	1873
Steinbeck	Cannery Row	1945
Wharton	Ethan Frome	1911
Conrad	Lord Jim	1900
Kerouac	Big Sur	1962

**Book**

The highlighted row refers to the book “*Cannery Row*” written by Steinbeck and first published in 1945.

## THE ROWS II

---

In formal language, a row is called a *tuple*.

A  $k$ -tuple is a mathematical object (similar to a vector) with  $k$  values in a specific order.

A 3-tuple stores a *triple* of values.

( Steinbeck , Cannery Row , 1945 )

## THE ATTRIBUTES

---

The **attributes** of an entity set is the collection of **defining properties** that identify an entity in that entity set.

For example, in the table above, an individual book is defined by its **author**, **title** and **date**.

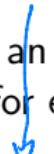
In other words, the attributes define **what type of data** is to be stored for each entity.

In the **Book** table, each row stores the author, title and date (and no other data) for a particular book.

## THE COLUMNS

---

Each column has a *name*, which is the name of an attribute (in the header), and stores the values of the attribute for each entity.



author	title	date
Tolstoy	Anna Karenina	1873
Steinbeck	Cannery Row	1945
Wharton	Ethan Frome	1911
Conrad	Lord Jim	1900
Kerouac	Big Sur	1962

For example, *date* is an *attribute*, while 1873 is one of the *values* of this attribute.

## A Student TABLE

---

Which of the following attributes would be needed to store student information for a university database?

Name, Height, Eye Colour, Nickname, Address, Date of Birth, Favourite Movie, **Student Number**, Gender, Weight, Emergency Contact, Citizenship/Visa Status

*most important*

# RELATIONS

---

The entire tabular structure is called a *table* or a *relation*<sup>2</sup>.

The *number of columns* of a table is called the *arity* of the table, and so the *Book* table has arity 3, while *Movie* has arity 5.

Two rows of a table are equal if they have the *same value* in every column.

In theory, a relational table should not have duplicate rows, but in actual SQL tables in a real database, duplicate rows are usually permitted.

---

<sup>2</sup>There is a mathematical reason for this name.

## TYPES

---

The entries of the **Book** table are ordered triples of the form

$$(author, name, date)$$

So

(**"Dickens"**, **"David Copperfield"**, 1850)



is a *legal tuple* for the relation **Book**.

DATA INTEGRITY

## TYPES II

---

integer not string.

Some tuples are *obviously incompatible* with the table structure.

- ▶ ("Dickens", 1850, "David Copperfield") ← *some SQL reject this*  
*Incorrect types*
- ▶ ("Dickens", "David Copperfield") ← *all SQL versions reject this*  
*Incorrect arity*

On the other hand,

("Dickens", "The Da Vinci Code", 2003) ←

is perfectly *legal*, although it is factually incorrect.

# WORKFLOW

---

In a working database, users will be entering, updating and deleting data from the tables throughout the lifetime of the database — books will be lent out and returned, stock will be bought and sold etc.



While it is very easy to deal with rapidly changing *data*, it is much harder to deal with changing *tables*.

In other words, the *structure* of the database is largely *unchanging*, although its *contents* are *frequently* changing.

The moral of this is to *carefully design* the database to ensure that every business requirement is captured.

CITS1402  
Relational Database Management Systems

Video 04 — SQL I

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Structured Query Language

---

*Structured Query Language*, or SQL, is the **standard computer language** used to interact with a relational database.

- ▶ A user connects to a DB, usually containing several tables
- ▶ The user writes a *SQL query*, which is a *declarative statement* giving the logical description of a new table *imperative*
- ▶ The RDBMS optimises the query, runs it, and produces a relational table containing the output
- ▶ The SQL query may, or may not, *alter* the existing tables

“Tables in, tables out — everything is a table”

# Structured Query Language

---

Structured Query Language or SQL is just an *ISO standard* specifying the *syntax* and *semantics* of a declarative language for accessing a relational database.

grammar      meaning

- ▶ Syntax — The *syntax* determines which statements are *legal expressions* in the language
- ▶ Semantics — The *semantics* determine *the meaning* of each of the legal expressions

## Many variants

---

There is no compulsion on any database vendor to stick precisely to the standard, and so there are numerous “*flavours*” of SQL.

Every *actual* database system *omits* some SQL commands, but *includes* some non-standard extensions.

Most SQL vendors implement the same set of “core features” of the standard, but you should not expect your SQL to be immediately transferable.



## Setting up your environment

---

### SQLite -

To use SQLite you just need the executable file which is

- ▶ pre-installed on Macs, and
- ▶ can be downloaded on Windows.

\$ sqlite3

Then use the Terminal / Command Prompt to

- ▶ create a folder for your CITS1402 work, and
- ▶ change into that folder, and
- ▶ run the SQLite program.

# Open SQLite

---

```
00013890@DEP52010 AFL % sqlite3
SQLite version 3.37.0 2021-12-09 01:34:53
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> - - -
```

In the *SQLite program*

# Dot commands

---

*Dot commands* are special SQLite commands that typically control aspects of how the user interacts with SQLite.

These are unique to SQLite and are not part of SQL.

sqlite> .help	
.auth ON OFF	Show authorizer callbacks
.backup ?DB? FILE	Backup DB (default "main") to FILE
.bail on off	Stop after hitting an error. Default OFF
.binary on off	Turn binary output on or off. Default OFF
.cd DIRECTORY	Change the working directory to DIRECTORY
.changes on off	Show number of rows changed by SQL
.check GLOB	Fail if output since .testcase does not match
.clone NEWDB	Clone data into NEWDB from the existing database
.databases	List names and files of attached databases
.dbconfig ?op? ?val?	List or change sqlite3_db_config() options
.dbinfo ?DB?	Show status information about the database
.dump ?TABLE? ...	Render all database content as SQL
.echo on off	Turn command echo on or off
.eqp on off full ...	Enable or disable automatic EXPLAIN QUERY PLAN
.excel	Display the output of next command in a spreadsheet
.exit ?CODE?	Exit this program with return-code CODE
.expert	EXPERIMENTAL. Suggest indexes for specified queries
.fullschema ?--indent?	Show schema and the content of sqlite_stat tables
.headers on off	Turn display of headers on or off

A very useful one is **.quit** which ends the session.

# Databases

---

Each database is stored as a *file* in the file system.

To do useful work you need to “attach” the sqlite3 program to the file containing the database.

```
00013890@DEP52010 AFL % sqlite3
SQLite version 3.37.0 2021-12-09 01:34:53
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open AFLResult.db
sqlite> .tables
AFLResult
sqlite>
```

*prepared earlier.*

The database is in `afl.db` and it contains *one table*, called `AFLResult`.

# What does this table contain?

This table lists the results of all the home-and-away Australian Football League (AFL) games from 2012–2021.

year	round	homeTeam	awayTeam	homeScore	awayScore
2012	1	GWS	Sydney	137	100
2012	1	Richmond	Carlton	81	125
2012	1	Hawthorn	Collingwood	137	115
2012	1	Melbourne	Brisbane Lions	178	119
2012	1	Gold Coast	Adelaide	68	137
2012	1	Fremantle	Geelong	105	101
2012	1	North Melbourne	Essendon	102	104
2012	1	Western Bulldogs	West Coast	187	136
2012	1	Port Adelaide	St Kilda	89	185
2012	2	Brisbane Lions	Carlton	63	154
2012	2	Essendon	Port Adelaide	111	186
2012	2	Sydney	Fremantle	94	81
2012	2	West Coast	Melbourne	166	158
2012	2	Collingwood	Richmond	85	164
2012	2	Adelaide	Western Bulldogs	82	164

GWS  
Richmond  
Hawthorn

## The columns of the table

---

In the AFL year, there are (about) 23 weekly rounds from April — September.

The attributes for each result are:

- ▶ The year and the round
- ▶ The two teams (home and away)
- ▶ The score for each team (home and away)

So this gives a table of *arity* 6, with each 6-tuple comprising 2 *integers*, 2 *strings* and 2 more *integers*.

# How are these represented?

---

*structure*

The *schema* of a table is the list of *attributes* (i.e., columns).

Each attribute has a *name* and a *type*.

```
sqlite> .schema AFLResult
CREATE TABLE AFLResult(
    year INT,
    round INT,
    homeTeam TEXT,
    awayTeam TEXT,
    homeScore INT,
    awayScore INT);
sqlite>
```

} the 'table creation' code

The name, attributes and types of the table, which collectively define the table's *structure* is called the *schema* of the table.

# Let's do some SQL

---

The most fundamental action is to *query* the database.

To do this we use the `SELECT` statement, which is the workhorse statement of SQL.

```
SELECT desired_columns  
FROM source_table ;
```

```
SELECT homeTeam FROM AFLResult;
```

## How does this work?

---

Each of the 1935 rows of the table `AFLResult` are processed.

For each row, the value of the column `homeTeam` is *selected*. ←

The result is a table with 1935 rows and 1 column, which will cause the output to overflow the terminal window.

```
western Bulldogs
Port Adelaide
Brisbane
Sydney
Melbourne
Geelong
Richmond
sqlite>
```

## Selecting more than one column

Just *list the names* of all the columns you want:

```
SELECT year, round, homeTeam, awayTeam  
FROM AFLResult;
```

(2021, 12, 'Gold Coast', 'Essendon') →

year	round	homeTeam	awayTeam
2021	22	Gold Coast	Essendon
2021	22	Fremantle	West Coast
2021	23	Western Bulldogs	Port Adelaide
2021	23	Richmond	Hawthorn
2021	23	Sydney	Gold Coast
2021	23	Brisbane Lions	West Coast
2021	23	Geelong	Melbourne
2021	23	Carlton	GWS
2021	23	St Kilda	Fremantle
2021	23	Essendon	Collingwood
2021	23	Adelaide	North Melbourne

sqlite> ↴ 2021/22/Gold

SQLite just uses the vertical bar | as a *column separator*<sup>1</sup>.

<sup>1</sup>The dot-command .separator can be used to change this

## More sophisticated queries

---

Usually we are interested in asking *more complicated* questions.

Let's ask the database to list the games that have ended in a *draw* — this is when the home team and the away team have the same score.

```
SELECT year, round, homeTeam, awayTeam  
FROM AFLResult  
WHERE homeScore = awayScore;
```

boolean

The **WHERE** clause *filters the rows* according to some *boolean condition*, and only keeps those that pass.

# The draws

---

```
sqlite> SELECT year, round, homeTeam, awayTeam  
...> FROM AFLResult WHERE homeScore = awayScore;  
2012|23|Richmond|Port Adelaide  
2013|18|Sydney|Fremantle  
2014|23|Carlton|Essendon  
2015|14|Adelaide|Geelong  
2015|18|Gold Coast|West Coast  
2015|21|St Kilda|Geelong  
2017|15|GWS|Geelong  
2017|16|Hawthorn|GWS  
2017|19|Collingwood|Adelaide  
2018|15|St Kilda|GWS  
2020|12|Collingwood|Richmond  
2020|11|Gold Coast|Essendon  
2021|13|North Melbourne|GWS  
2021|18|Melbourne|Hawthorn  
2021|23|Richmond|Hawthorn
```

As SQL processes *each row*, it uses the values *for that row* to

- ▶ decide whether the boolean condition is satisfied, and
- ▶ what row to add to the output table if it is.

## A fuller version of SELECT

---

Gradually we'll add more and more functionality to the `SELECT` statement, until we understand all of the following features:

```
SELECT columns  
FROM tables  
WHERE rowconditions  
GROUP BY groups  
HAVING groupconditions  
ORDER BY sortcolumns  
LIMIT numrows;
```

# A row-processing machine

---

A SQL statement using just `SELECT / FROM / WHERE` just processes every row of the input table *once*.

For each row of input, SQL checks

- ▶ Does it satisfy the `WHERE` condition?
  - ▶ If *No*, then SQL discards the row and moves to the next one
  - ▶ If *Yes*, then SQL forms a new row by keeping the column(s) named after `SELECT`

There is no (obvious) way to *compare* two rows of a table.

So how, for example, could we find the highest-scoring games?

CITS1402  
Relational Database Management Systems

Video 05 — RDBMS

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## The RDBMS model

---

Many different vendors sell *implementations* of SQL.

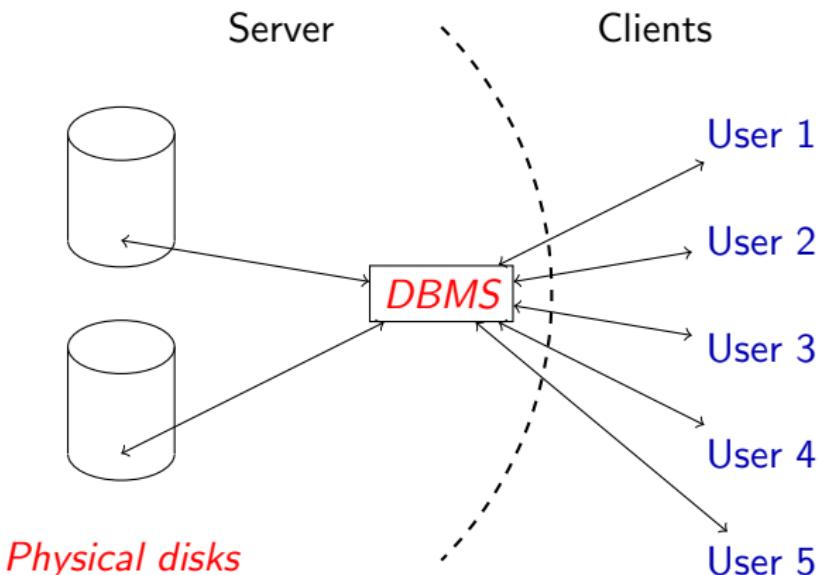
The most commonly used (Oracle, MySQL, PostgreSQL and Microsoft SQL Server) all use the *client/server* model.

This means that a *server* is running on a computer, while “listening” for connection requests from *clients*.

The *clients* are individual (people or computers) who connect to the server and then issue SQL queries.

# The DBMS client/server model

---



## What does a DBMS provide?

---

There is a free online course “*Introduction to Databases*” presented by database guru, Jennifer Widom, from Stanford University.

<https://www.edx.org/course/databases-1-introduction-to-relational-databases>

In this, she describes a DBMS as providing “*efficient, reliable, convenient, and safe multi-user storage of – and access to – massive amounts of persistent data*”.

(In general, some of the video lectures and supporting materials from this course are a useful resource for this unit.)

## In more detail

---

More precisely, relational databases, provide many benefits to the data user, in particular features such as:

- ▶ Data Independence
- ▶ Efficiency
- ▶ Data Integrity
- ▶ Data Administration
- ▶ Concurrency Control
- ▶ Application Development

(These benefits are not restricted to relational databases only—each type of database does some things well and other things less well.)

# Data Independence I

---

Data independence provides analogous benefits to the *encapsulation* found in object-oriented programming languages:

- ▶ Users and applications use a *logical model* of the underlying data, rather than directly manipulating the physical files storing the data.
- ▶ *Implementation* of physical storage can be altered or improved without affecting client code.
- ▶ Physical storage can be *remote*, or distributed, or both, with no alteration in client code.

## Data Independence II

---

Users *query* an RDBMS, using something like the following<sup>1</sup>.

```
SELECT name  
FROM Student  
WHERE snum = 22041020;
```

This a *declarative* rather than *imperative* statement.

User does not need to know *where* or *how* the data is stored; this is all delegated to the RDBMS.

---

<sup>1</sup>Do not worry if you do not understand this yet!

## Efficiency

---

An RDBMS implements storage and retrieval strategies to make the most common operations as fast as possible.

An RDBMS maintains various *indexes* to the data → fast search.

Given a complex query, the RDBMS will devise a *query execution plan* to answer the query.

Database storage and indexing strategies are extremely sophisticated applications of data structures techniques.

# Data Integrity

---

An RDBMS keeps the database in a *consistent state* by enforcing *integrity constraints* derived from relevant “business rules”.

Changes often have a *ripple effect* of consequences.

- ▶ *If a discontinued product is deleted from a catalogue, then any bundles including it should also be deleted.*

An RDBMS manages recovery from unexpected interruptions, such as power cuts or communications breakdowns.

# Data Administration

---

A multi-user RDBMS allows the organization a fine *degree of control* over who is permitted various levels of access to the database.

A multi-user RDBMS permits arbitrarily fine-grained control, allowing different users to have different *views* of the same underlying data.

For example, a lecturer may be able to look up a student's academic record, but not their personal or financial details, while only certain staff will be able to *alter* their academic record.

## Concurrency Control

---

In a large organization, there will often be several people accessing the same data item at the same time.

While this is not a problem if all users are simply *viewing* the data, it becomes a major problem if some of the users need to *update* the data.

For example, an airline reservation system may have several travel agents viewing availability at the same time, but the DBMS must prevent two agents from booking the same seat at the same time.

# Application Development

---

*Analysing* or *presenting* data may need more sophisticated tools.

Many general purpose programming languages (Python, Java, C, R, PHP) can directly access a DBMS through *connectors* which provide standardised interfaces to connect to and query the database.

The power and success of this form of application development can be seen by the fact that essentially every large dynamic website is backed by a relational database.

# SQLite

---

The SQLite tagline is:

*Small. Fast. Reliable. Choose any three.*

Dealing with potential simultaneous users adds *significant complication and overhead* to an implementation of SQL.

But the vast majority of databases (in number, not size) never have more than one user at a time.

With SQLite,

- ▶ A database is stored as a *single file* in the user's file system
- ▶ Users *directly interact* with the database using the command-line program `sqlite3`

# The command-line interface

---

## The user

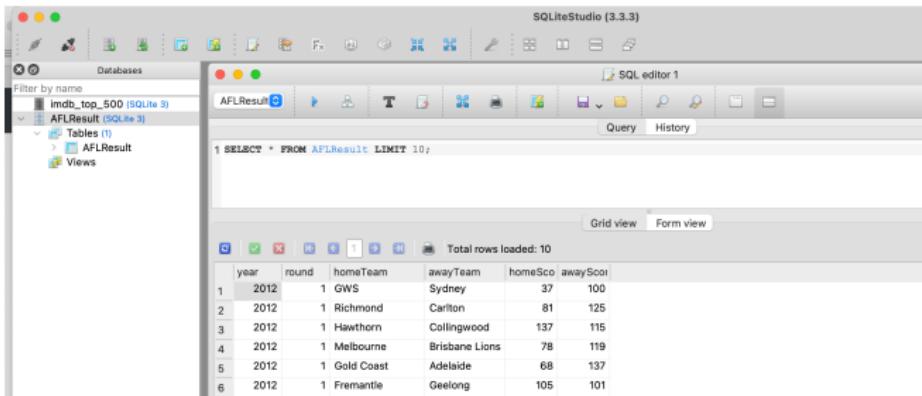
- ▶ *Starts* sqlite3 session in a *terminal window*
- ▶ *Attaches* a database file using a *dot command*
- ▶ Repeatedly *types* either SQL or dot commands
- ▶ *Quits* the session with .quit

```
00013890@DEP52010 AFL % sqlite3
SQLite version 3.37.0 2021-12-09 01:34:53
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open AFLResult.db
sqlite> .tables
AFLResult
sqlite> SELECT * FROM AFLResult LIMIT 5;
2012|1|GWS|Sydney|37|100
2012|1|Richmond|Carlton|81|125
2012|1|Hawthorn|Collingwood|137|115
2012|1|Melbourne|Brisbane Lions|78|119
2012|1|Gold Coast|Adelaide|68|137
sqlite> ;
```

# SQL Studio

---

Can also use a *graphical user interface* (or GUI).



SQLiteStudio is a free GUI that runs on Win, Mac and Linux.

Download from <https://sqlitestudio.pl>.

# CITS1402

## Relational Database Management Systems

### Video 06 — JOIN

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# A bad design

---

Imagine a spreadsheet storing *geographical* data.

Continent	Country	Capital	City	Population
Europe	England	London	London	7285000
Europe	England	London	Manchester	430000
Europe	France	Paris	Paris	2125246
Europe	France	Paris	Montpellier	225392
Asia	Thailand	Bangkok	Bangkok	6320174
Asia	Thailand	Bangkok	Chiang Mai	171100

This is a bad design for storing data.

## A single table

---

The data structure is a single *table*, where:

- ▶ Each *row* stores all the data for *one city*
- ▶ Each *column* stores the *values* of a single *attribute*

There is nothing wrong *a priori* with using a table.

But *this* design combines both *redundancy* and *fragility*.

## Three anomalies

---

An *anomaly* is an error or inconsistency in the database.

- ▶ **Insertion Anomaly**

*Inserting* a row may require irrelevant information

—Need to know a country's capital to enter a new city.

- ▶ **Deletion Anomaly**

*Deleting* a row risks losing information

—Deleting the last city loses all the country/capital information.

- ▶ **Update Anomaly**

*Updating* a row risks leaving the database inconsistent

—Changing the capital's name alters multiple rows.

## The problem

---

Each row keeps track of *two distinct concepts*:

- ▶ Information about *cities*, and
- ▶ Information about *countries*.

These two concepts have very different *granularity*<sup>1</sup>.

The solution is to *split the information* into more than one table, so each table keeps track of data related to one logical concept

- ➡ This process is called *normalization* and will be covered later.

---

<sup>1</sup>level of detail

# Cities and Countries

---

Separate tables for “country-level” data and “city-level” data.

Continent	Country	Capital
Europe	England	London
Europe	France	Paris
Asia	Thailand	Bangkok

City	Population
London	7285000
Manchester	430000
Paris	2125246
Montpellier	225392
Bangkok	6320174
Chiang Mai	171100

This reduces *redundancy*, but now we cannot answer questions that require *both* country *and* city data:

*In which continent does the city of Ezeiza lie?*

# Connect the two tables

---

The tables must be *connected* usually through a *shared attribute*.

Continent	Country	Capital
Europe	England	London
Europe	France	Paris
Asia	Thailand	Bangkok

Country	City	Population
England	London	7285000
England	Manchester	430000
France	Paris	2125246
France	Montpellier	225392
Thailand	Bangkok	6320174
Thailand	Chiang Mai	171100

- An attribute of one table that refers to an attribute of a different table is called a *foreign key*.

# The world

---

We'll use a *simplified version* of a sample database called `world` (originally created by/for MySQL).

(The data is very old, but it is one of a handful of “classic” sample databases that have been used by generations of students.)

```
00013890@DEP52010 worldDatabase % sqlite3 sqliteWorld.db
SQLite version 3.37.0 2021-12-09 01:34:53
Enter ".help" for usage hints.
sqlite> .tables
City          Country          CountryLanguage
sqlite> 
```

Note: SQLite is not case-sensitive, so you can capitalise words or not, according to preference.

# The city table

---

```
sqlite> .schema City
CREATE TABLE City (
    id INT,
    name TEXT,
    countryCode TEXT,
    population INT );
```

# The cities

---

```
sqlite> .headers on
sqlite> SELECT * FROM City LIMIT 10;
id|name|countryCode|population
1|Kabul|AFG|1780000
2|Qandahar|AFG|237500
3|Herat|AFG|186800
4|Mazar-e-Sharif|AFG|127800
5|Amsterdam|NLD|1731200
6|Rotterdam|NLD|1593321
7|Haag|NLD|1440900
8|Utrecht|NLD|1234323
9|Eindhoven|NLD|1201843
10|Tilburg|NLD|193238
```

# The countries

---

```
sqlite> .schema Country
CREATE TABLE Country (
    code TEXT,
    name TEXT,
    capital INT,
    continent TEXT );
sqlite> SELECT * FROM Country LIMIT 10;
code|name|capital|continent
ABW|Aruba|129|North America
AFG|Afghanistan|11|Asia
AGO|Angola|56|Africa
AIA|Anguilla|62|North America
ALB|Albania|34|Europe
AND|Andorra|55|Europe
ANT|Netherlands Antilles|33|North America
ARE|United Arab Emirates|65|Asia
ARG|Argentina|69|South America
ARM|Armenia|126|Asia
sqlite>
```

# The languages

---

```
sqlite> SELECT * FROM CountryLanguage LIMIT 10;  
countryCode|language|isOfficial|percentage  
ABW|Dutch|T|5.3  
ABW|English|F|19.5  
ABW|Papiamento|F|76.7  
ABW|Spanish|F|7.4  
AFG|Balochi|F|0.9  
AFG|Dari|T|32.1  
AFG|Pashto|T|52.4  
AFG|Turkmenian|F|11.9  
AFG|Uzbek|F|8.8  
AGO|Ambo|F|2.4  
sqlite>
```

# Connecting the tables

---

Each country has a three-letter **code** in the table **Country**.

code	name
AUS	Australia
CHN	China
IDN	Indonesia
MYS	Malaysia
SGP	Singapore

This code is enough to *uniquely identify* the country <sup>2</sup>.

**DEFINITION** An attribute is called a **key** if it is impossible for different rows to have the *same value* of this attribute.

---

<sup>2</sup>There is an international standard assigning a three-letter code to each country

## Brief digression on keys

---

Another way to view a key is that the *value* of a key identifies *exactly one row*. For example,

- ▶ *Student Number* in a university's database
- ▶ *Account Number* in a bank's database
- ▶ *Tax File Number* in the ATO's database

Attributes like “*Name*”, “*Address*”, “*Date-of-Birth*” are usually *not* keys, because two people can have the same name, etc.

You *cannot tell* which attributes are keys by examining the rows that are currently in the table.

# CITS1402

## Relational Database Management Systems

### Video 07 — JOIN

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## Joining tables

---

The data has been *split* between tables so that each stores one type of information:

```
City( id, name, countryCode, population )  
Country ( code, name, capital, continent )
```

However the tables are *related* through special attributes known as *foreign keys*.

- ▶ The attribute `City.countryCode` is a *reference* to `Country.code`
- ▶ The attribute `Country.capital` is a *reference* to `City.id`

# What is the capital of Australia?

---

This query needs information from two tables

- ▶ Find the `id` of Australia's capital from `Country`
- ▶ Look up the `name` of Australia's capital from `City`

```
sqlite> SELECT capital FROM Country WHERE name = 'Australia';
135
sqlite> SELECT name FROM City WHERE id = 135;
Canberra
sqlite>
```

## Too cumbersome

---

This is far too *cumbersome* — we'd rather live with the redundancy than having to do queries in multiple steps!

This is where the ability to `JOIN` tables together is important.

- ▶ (Normalization) Big table is *normalized* into multiple small tables
- ▶ (Joining) Small tables are *joined* to reproduce big table

# Where is Ezeiza?

---

The information connecting cities to countries is in the table **City**.

```
SELECT countryCode  
FROM City  
WHERE name = "Ezeiza";
```

The output from this is **ARG**.

```
SELECT Continent  
FROM Country  
WHERE Code = "ARG";
```

The output from this is **South America**.

## Do it in one command

---

```
SELECT continent
FROM City, Country
WHERE City.countryCode = Country.code
AND City.name = "Ezeiza";
```

```
sqlite> SELECT continent
...> FROM City, Country
...> WHERE City.countryCode = Country.code
...> AND City.name = 'Ezeiza';
South America
sqlite>
```

# Two tables

---

The `FROM` statement lists *two tables* — what does this mean?

```
sqlite> .headers on
sqlite> SELECT * FROM City, Country LIMIT 20;
id|name|countryCode|population|code|name|capital|continent
1|Kabul|AFG|1780000|ABW|Aruba|129|North America
1|Kabul|AFG|1780000|AFG|Afghanistan|1|Asia
1|Kabul|AFG|1780000|AGO|Angola|56|Africa
1|Kabul|AFG|1780000|AIA|Anguilla|62|North America
1|Kabul|AFG|1780000|ALB|Albania|34|Europe
1|Kabul|AFG|1780000|AND|Andorra|55|Europe
1|Kabul|AFG|1780000|ANT|Netherlands Antilles|33|North America
1|Kabul|AFG|1780000|ARE|United Arab Emirates|65|Asia
1|Kabul|AFG|1780000|ARG|Argentina|69|South America
1|Kabul|AFG|1780000|ARM|Armenia|126|Asia
1|Kabul|AFG|1780000|ASM|American Samoa|54|Oceania
1|Kabul|AFG|1780000|ATA|Antarctica|Antarctica
1|Kabul|AFG|1780000|ATF|French Southern territories|Antarctica
1|Kabul|AFG|1780000|ATG|Antigua and Barbuda|63|North America
1|Kabul|AFG|1780000|AUS|Australia|135|Oceania
1|Kabul|AFG|1780000|AUT|Austria|1523|Europe
1|Kabul|AFG|1780000|AZE|Azerbaijan|144|Asia
1|Kabul|AFG|1780000|BDI|Burundi|552|Africa
1|Kabul|AFG|1780000|BEL|Belgium|179|Europe
1|Kabul|AFG|1780000|BEN|Benin|187|Africa
1|Kabul|AFG|1780000|BIL|Bolivia|11|South America
```

## The Cartesian product

---

The expression `City, Country` asks SQL to form the *full Cartesian product* of the two tables.

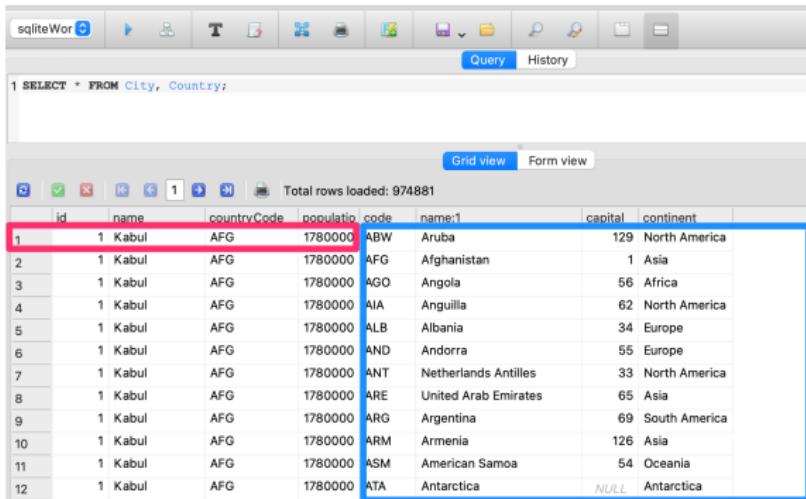
The rows of the `City, Country` each consist of a row from `City` “glued together” with a row from `Country` *in every way possible*.

There are 239 cities and 4079 countries, so this “product table” has 974881 rows.

# But it's (almost) all junk

---

But almost every row is the *deformed offspring* of some information about a city and information about an *unrelated* country.



The screenshot shows a screenshot of the sqliteWorm application interface. At the top, there is a toolbar with various icons. Below the toolbar, a tab bar has 'Query' selected. The main area contains a SQL query: '1 SELECT \* FROM City, Country;'. Below the query, a message says 'Total rows loaded: 974881'. A table is displayed with the following data:

	id	name	countryCode	population	code	name1	capital	continent
1	1	Kabul	AFG	1780000	ABW	Aruba	129	North America
2	1	Kabul	AFG	1780000	AFG	Afghanistan	1	Asia
3	1	Kabul	AFG	1780000	AGO	Angola	56	Africa
4	1	Kabul	AFG	1780000	AIA	Anguilla	62	North America
5	1	Kabul	AFG	1780000	ALB	Albania	34	Europe
6	1	Kabul	AFG	1780000	AND	Andorra	55	Europe
7	1	Kabul	AFG	1780000	ANT	Netherlands Antilles	33	North America
8	1	Kabul	AFG	1780000	ARE	United Arab Emirates	65	Asia
9	1	Kabul	AFG	1780000	ARG	Argentina	69	South America
10	1	Kabul	AFG	1780000	ARM	Armenia	126	Asia
11	1	Kabul	AFG	1780000	ASM	American Samoa	54	Oceania
12	1	Kabul	AFG	1780000	ATA	Antarctica	NULL	Antarctica

# But not quite

---

A few rows “happen” to have their first- and second-halves referring to the same country and so these are the useful rows.

Query History

```
1 SELECT * FROM City, Country;
```

Total rows loaded: 974881

	id	name	countryCode	populatio	code	name:1	capital	continent
1	1	Kabul	AFG	1780000	ABW	Aruba	129	North America
2	1	Kabul	AFG	1780000	AFG	Afghanistan	1	Asia
3	1	Kabul	AFG	1780000	AGO	Angola	56	Africa
4	1	Kabul	AFG	1780000	AIA	Anguilla	62	North America
5	1	Kabul	AFG	1780000	ALB	Albania	34	Europe
6	1	Kabul	AFG	1780000	AND	Andorra	55	Europe
7	1	Kabul	AFG	1780000	ANT	Netherlands Antilles	33	North America
8	1	Kabul	AFG	1780000	ARE	United Arab Emirates	65	Asia
9	1	Kabul	AFG	1780000	ARG	Argentina	69	South America

## Extract the useful tuples

---

How do we get the useful tuples?

```
SELECT *
FROM City, Country
WHERE City.countryCode = Country.code;
```

This operation is known as a **JOIN** of two tables.

The condition that tells SQL when one of these “double-width” rows is valid is called the *join condition*.

# The JOIN

Query History

```
1 SELECT * FROM City, Country
2 WHERE City.countryCode = Country.code;
```

Total rows loaded: 4079

	id	name	countryCode	population	code	name:1	capital	continent
1	1	Kabul	AFG	1780000	AFG	Afghanistan	1	Asia
2	2	Qandahar	AFG	237500	AFG	Afghanistan	1	Asia
3	3	Herat	AFG	186800	AFG	Afghanistan	1	Asia
4	4	Mazar-e-Sharif	AFG	127800	AFG	Afghanistan	1	Asia
5	5	Amsterdam	NLD	731200	NLD	Netherlands	5	Europe
6	6	Rotterdam	NLD	593321	NLD	Netherlands	5	Europe
7	7	Haag	NLD	440900	NLD	Netherlands	5	Europe
8	8	Utrecht	NLD	234323	NLD	Netherlands	5	Europe

## Logical vs practical

---

This is a *logical description* only of the join process

In practice, the system will not *actually* create 974881 rows and then throw away all but 4070 of them.

But SQL will *examine* the query, and then choose some efficient way to actually *implement* it.

This permits the user to think only about what they want from *the data*, leaving the organisation of the computation to the system.

# Names

---

The attributes of **City**, **Country** are

(**id**, **name**, **countryCode**, **population**, **code**, **name**, **capital**,  
**continent**)

Query History

```
1 SELECT * FROM City, Country
2 WHERE City.countryCode = Country.code;
```

Total rows loaded: 4079

Grid view Form view

	<b>id</b>	<b>name</b>	<b>countryCode</b>	<b>population</b>	<b>code</b>	<b>name:1</b>	<b>capital</b>	<b>continent</b>
1	1	Kabul	AFG	1780000	AFG	Afghanistan	1	Asia
2	2	Qandahar	AFG	237500	AFG	Afghanistan	1	Asia
3	3	Herat	AFG	186800	AFG	Afghanistan	1	Asia
4	4	Mazar-e-Sharif	AFG	127800	AFG	Afghanistan	1	Asia
5	5	Amsterdam	NLD	731200	NLD	Netherlands	5	Europe
6	6	Rotterdam	NLD	593321	NLD	Netherlands	5	Europe
7	7	Haag	NLD	440900	NLD	Netherlands	5	Europe
8	8	Utrecht	NLD	234323	NLD	Netherlands	5	Europe

But what if I just want the city and country *names*?

## Disambiguation

---

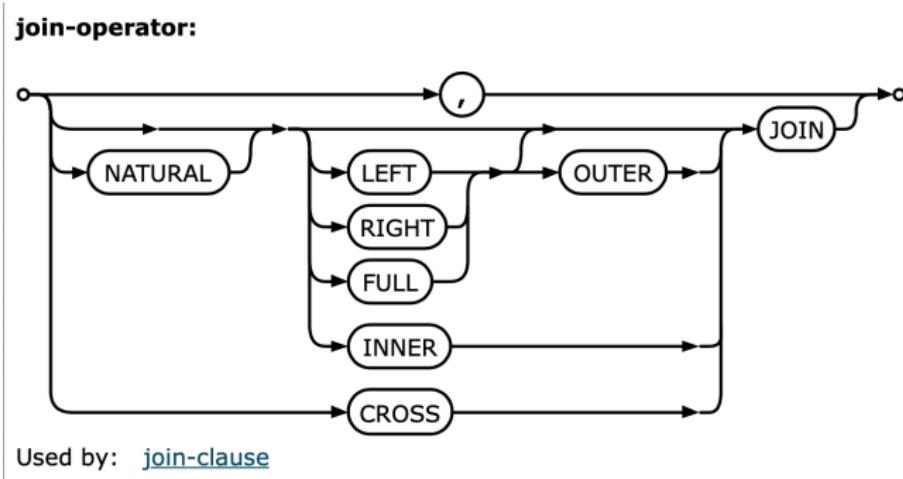
We have to distinguish between the **name** that came from **City** and the ***other name*** that originally came from **Country**.

```
SELECT City.name, Country.name  
FROM City, Country  
WHERE countryCode = code;
```

# Too many ways

---

SQL has *many different ways* to accomplish the same thing, as shown in this *syntax diagram* from sqlite.org.



## One more example

---

QUESTION Write a query to produce a table containing the *name* of each city and the *continent* in which it lies.

1. Identify *which tables* contain this information

City names are in *City* while continent names are in *Country*

2. Work out *how to join* the tables correctly

Use the join condition *City.countryCode = Country.code*

3. Decide which columns to *extract* from the joined table

We want *City.name* and *Country.continent*

# CITS1402

## Relational Database Management Systems

### Video 08 — DDL

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Data Definition Language

---

Traditionally, SQL commands are divided into *groups* according to their purpose

- ▶ DDL — Data Definition Language

Command used to *define the structure* of the database, such as the tables, the attributes and their types (and more advanced structural properties).

- ▶ DML — Data Manipulation Language

Commands used to *manipulate the data* in the database, including the CRUD operations

# DDL commands

---

- ▶ **CREATE TABLE**  
*Create a new (empty) table*
- ▶ **DROP TABLE**  
*Delete a table (and all its data)*
- ▶ **ALTER TABLE**  
*Rename (a table or column), add a column or drop a column*

## A student database

---

To illustrate the DDL and for future use, we'll build a toy database to store student marks, according to the following requirements:

*A student has a name and a unique student number. A unit has a name and a unique unit code. Students enrol in units in a particular year and on completion they get a mark for that unit. The mark for any unit is an integer between 0 and 100 inclusive, while the pass mark is 50. A student may take a unit in multiple years.*

# The tables

---

We'll use three tables

- ▶ **Student**

Student name and student number (key)

- ▶ **Unit**

Unit name and unit code (key)

- ▶ **Enrolment**

Student number, unit code, year and mark

Why do year and mark belong to **Enrolment**?

## Create the tables

---

```
CREATE TABLE Student (
    sNum INTEGER,
    sName TEXT );

CREATE TABLE Unit (
    uCode TEXT,
    uName TEXT );

CREATE TABLE Enrolment (
    sNum INTEGER,
    uCode TEXT,
    yr INTEGER,
    mark INTEGER
) ;
```

## Inserting new rows

---

This is done with the DML statement `INSERT INTO`

```
INSERT INTO Student VALUES(1, "Amy");
INSERT INTO Student VALUES(2, "Bai");
```

The words in **blue** are SQL keywords and all of them must be present in exactly the right order.

In most implementations of SQL, the **values** must be of the *correct type*, but SQLite treats types in its own unique fashion.

## Inserting new rows

---

```
sqlite> CREATE TABLE Student ( sNum INTEGER,
...>     sName TEXT );
sqlite> SELECT * FROM Student;
sqlite> INSERT INTO Student VALUES (1,"Amy");
sqlite> SELECT * FROM Student;
1|Amy
sqlite> INSERT INTO Student VALUES(2,"Bai");
sqlite> SELECT * FROM Student;
1|Amy
2|Bai
sqlite>
```

## Add some rows to Unit

---

```
INSERT INTO Unit VALUES ('CITS1402', 'RDBMS');
INSERT INTO Unit VALUES ('CITS1401', 'Python');
INSERT INTO Unit VALUES ('MATH1001', 'Calculus');
INSERT INTO Unit VALUES ('MATH1002', 'Algebra');
```

Note: SQLite doesn't care if you use 'CITS1402' or "CITS1402".

But what if a unit is called *Einstein's Theory of Relativity*?

## Some enrolments

---

```
INSERT INTO Enrolment VALUES(1, 'CITS1402', 2020, 55);
INSERT INTO Enrolment VALUES(1, 'MATH1001', 2021, 80);
INSERT INTO Enrolment VALUES(2, 'CITS1402', 2021, 42);
```

Bai wants to repeat CITS1402 in 2022 — can we enter this enrolment into the database while his mark is not yet known?

## The special value `NULL`

---

`NULL` is used to represent values that are *unknown* or *undefined*.

```
INSERT INTO Enrolment VALUES(2, "CITS1402", 2022, NULL);
```

From the `sqlite3` prompt a `NULL` field just appears blank (this can be changed with the dot command `.nullvalue`).

```
sqlite> SELECT * FROM Enrolment;
1|CITS1402|2020|55
1|MATH1001|2021|80
2|CITS1402|2021|42
2|CITS1402|2022|
sqlite>
```

## Back to the DDL

---

Let's make the tables *more robust* to data entry errors.

The first step is to tell SQLite when a field is a *key* – this prevents a whole range of “fat-finger” data-entry errors.

```
CREATE TABLE Student (
    sNum INTEGER PRIMARY KEY,
    sName TEXT );

CREATE TABLE Unit (
    uCode TEXT PRIMARY KEY,
    uName TEXT );
```

So cannot have *two students* with the same sNum.

# Protection for your tables

---

The fact that `sNum` is declared to be a key is a decision made by the database designer.

This is driven by the *business logic* saying that in the university, a student number *uniquely identifies* a student.

If SQL is *explicitly told* that `sNum` is a key, then it will stop duplicate values being created in that column.

```
sqlite> CREATE TABLE Student (snum INTEGER PRIMARY KEY, name TEXT);
sqlite> INSERT INTO Student VALUES (1, 'Amy');
sqlite> INSERT INTO Student VALUES (1, 'Bai');
Error: stepping, UNIQUE constraint failed: Student.snum (19)
sqlite>
```

## When fields can't be NULL

---

Based on the user requirements, the database designer may decide that a particular field should never take the value `NULL`.

For example, although two students may have the *same* name, it is reasonable to insist that a student always has *some* name.

```
CREATE TABLE Student (
    sNum INTEGER PRIMARY KEY,
    sName TEXT NOT NULL );
```

```
sqlite> INSERT INTO Student VALUES(6,NULL);
Error: NOT NULL constraint failed: Student.sName
sqlite>
```

## Altering the table

---

SQLite supports the most fundamental table alteration commands:

- ▶ Changing the name of the table
- ▶ Changing the name of a column
- ▶ Adding a column
- ▶ Dropping a column (only since version 3.35)

```
ALTER TABLE Student  
    ADD COLUMN email TEXT;
```

```
ALTER TABLE AFLResult  
    DROP COLUMN round;
```

## The DDL statements

---

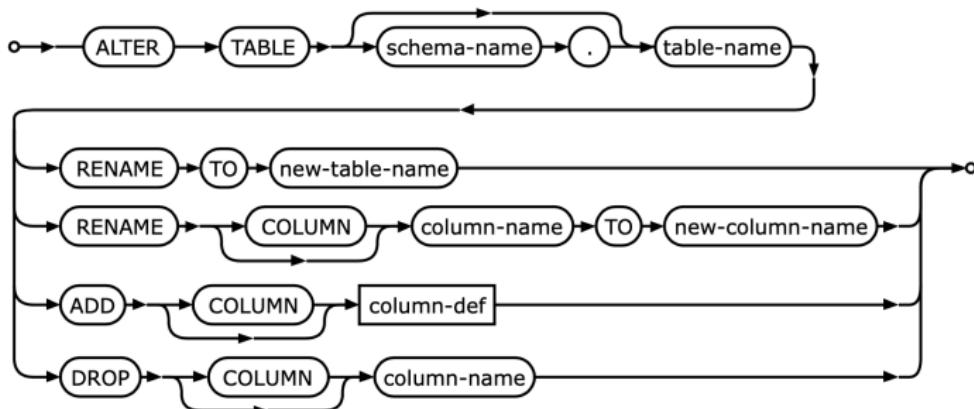
The major DDL statements all contain the word **TABLE**.

```
CREATE TABLE tablename ...
DROP TABLE tablename ...
ALTER TABLE tablename ...
```

# Reading the docs

---

## alter-table-stmt:



## column-def:

CITS1402  
Relational Database Management Systems

Video 09 — SELECT

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## More properties of SELECT

---

Today's lecture explores the `SELECT` statement in more detail

Recall the basic structure:

- ▶ `SELECT`  
Chooses *which columns* are to be returned by the query
- ▶ `FROM`  
Constructs the *initial table* to pass to `WHERE`
- ▶ `WHERE`  
Selects the *rows* that are to be passed to `SELECT`

# Which students are doing CITS1402?

---

Suppose we have a simple marks database:

`Student(sNum, sName, email)`

`Enrolment(sNum, uCode, yr, mark)`

`Unit(uCode, uName)`

*Write a SQL query to list the names and emails of the students enrolled in CITS1402 in 2022.*

## Make a plan

---

- ▶ We need to use **Student** and **Enrolment**  
*Names and email addresses are in Student and the enrolment details in Enrolment*
- ▶ **Join** the tables based on **student number**  
*Need to throw out rows where the student data and enrolment data are about different students*
- ▶ **Keep** only rows about students in CITS1402 in 2022.  
*Other rows are irrelevant to this question*
- ▶ **Choose** just the **name** and **email** for these students  
*Make sure the query provides exactly what is required*

## Another way to JOIN

---

Last week we saw that we can **JOIN** tables

```
SELECT *
FROM Student, Enrolment
WHERE Student.sNum = Enrolment.sNum;
```

Another way to express this is

```
SELECT *
FROM Student JOIN Enrolment
ON Student.sNum = Enrolment.sNum;
```

This formulation of **JOIN** highlights the *join condition*.

## The output of the JOIN

---

```
sqlite> .open marks.db
sqlite> .headers on
sqlite> .mode column
sqlite> .width 8 8 8 8 8 8 8 8
sqlite> SELECT *
      ...> FROM Student JOIN Enrolment
      ...> ON Student.sNum = Enrolment.sNum;
    sNum      sName      email        sNum      uCode      yr      mark
    -----  -----  -----  -----  -----  -----  -----
    1        Amy      amy@gmail.com  1        CITS1402  2020      55
    1        Amy      amy@gmail.com  1        MATH1001  2021      80
    2        Bai      bai@qq.com   2        CITS1402  2021      42
    2        Bai      bai@qq.com   2        CITS1402  2022
```

## A sample query

---

Let's use this in the full query:

```
SELECT ....  
FROM Student JOIN Enrolment  
ON Student.sNum = Enrolment.sNum  
WHERE ...
```

## Which rows to keep?

---

The `WHERE` clause defines a *boolean expression* involving some or all of the following:

- ▶ The *columns* of the source table
- ▶ *Arithmetic operators* and/or *functions*
- ▶ *Constant* values (numbers or text)
- ▶ *Boolean operators* like `AND`, `OR`, `NOT` and so on.

This expression is evaluated for *every row*, substituting the values in the named columns.

If the expression is false, then the row is discarded, else it is passed to the `SELECT` statement for further processing.

## WHERE filters

---

Only keep the rows relating to CITS1402 in 2022:

```
uCode = 'CITS1402' AND yr = 2022
```

```
sqlite> SELECT *  
...> FROM Student JOIN Enrolment  
...> ON Student.sNum = Enrolment.sNum  
...> WHERE uCode = 'CITS1402' AND yr = 2022;  
sNum      sName     email      sNum      uCode      yr      mark  
-----  -----  -----  -----  -----  -----  -----  
2          Bai      bai@qq.com  2        CITS1402  2022  
sqlite>
```

## SELECT chooses columns

---

```
SELECT sName, email
FROM Student JOIN Enrolment
ON Student.sNum = Enrolment.sNum
WHERE uCode = 'CITS1402' AND yr = 2022;
```

```
sqlite> SELECT sName, email
...> FROM Student JOIN Enrolment
...> ON Student.sNum = Enrolment.sNum
...> WHERE uCode = 'CITS1402' AND yr = 2022;
sName      email
-----
Bai        bai@qq.com
sqlite>
```

# A query in action

---

sNum	sName	email	sNum	uCode	yr	mark
1	Amy	amy@gmail.com	1	CITS1402	2020	55
1	Amy	amy@gmail.com	1	MATH1001	2021	80
2	Bai	bai@qq.com	2	CITS1402	2021	42
2	Bai	bai@qq.com	2	CITS1402	2022	

# A query in action

---

sNum	sName	email	sNum	uCode	yr	mark
2	Bai	bai@qq.com	2	CITS1402	2022	

# A query in action

---

	sName	email				
	Bai	bai@qq.com				

## Slice and Dice

---

Another way to remember this is that in a SQL query:

- ▶ `FROM` constructs an initial table,
- ▶ `WHERE` *slices* it horizontally, removing *rows*
- ▶ `SELECT` *dices* it vertically, removing *columns*

The surviving rows and columns are the output table.

There are many variations and extensions, but this is really the foundation of how SQL works.

## It's all tables

---

The *output* of this SQL query is a *table* — it has *named columns* and rows of values, just like a *stored table*.

So far, the `FROM` statement has just used table *names*.

```
SELECT * FROM Student, Enrolment  
WHERE ..
```

Later we'll see that a table *name* can be replaced by a table *expression* — in other words, a SQL query.

# Think like a machine

---

A SQL query will construct the table *exactly* as you tell it to.

Sometimes (perhaps often) this is not what you *want*.

To debug SQL queries, you need to “think like the machine”

- ▶ What is the *initial* table?
- ▶ What *rows* does the **WHERE** condition discard?
- ▶ What *columns* does the **SELECT** statement keep?

# CITS1402

## Relational Database Management Systems

### Video 10 — Types

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Types

---

When a table is created with `CREATE TABLE` the database designer has to specify the *name* and *type* of each column.

The type indicates (to SQL) what *kind of data* will be stored in that column — numbers, strings, boolean values etc.

This information can be used to assist SQL in various ways:

- ▶ Efficient choice of data structures when storing and accessing tables
- ▶ Improved data integrity by ensuring all operations involve compatible types

## A unit code

---

At UWA, unit codes (such as 'CITS1402') are always *exactly* 8 characters long.

According to the SQL standard, the most appropriate way to declare this field would be:

```
CREATE TABLE Unit(
    unitCode CHAR(8),
    ...
);
```

The system can then store this column with no wasted space.

## A unit name

---

On the other hand, the *name* of a UWA unit does not have a standard length:

```
'Relational Database Management Systems'  
'Network Science'
```

Maybe the length is limited to, say, 64 characters.

```
CREATE TABLE Unit(  
    unitCode CHAR(8),  
    unitName VARCHAR(64),  
    ...  
);
```

Here **VARCHAR** means “variable-length string of characters”.

## Dealing with VARCHAR

---

The system now has a choice:

- ▶ Waste space

*Allocate exactly 64 bytes for each unit name, even if it is much shorter than 64 characters, but get very rapid access.*

- ▶ Waste time

*Allocate precisely the right number of characters for each name, but consume more time locating each value.*

When SQL was young, both disk space and computer time were scarce, and so this *fine-grained control* over low-level issues was important.

## A unit description

---

Suppose we now wish to add the *unit description* which may be several paragraphs long.

```
CREATE TABLE Unit(
    unitCode CHAR(8),
    unitName VARCHAR(64),
    unitDescription TEXT
);
```

Here **TEXT** means an arbitrarily long sequence of characters which in CS is usually called a *string*.

## Same for numbers

---

Similarly there is very fine control over the storage of *integers* — for example MySQL allows the following integer types.

- INTEGER
- INT
- TINYINT
- MEDIUMINT
- BIGINT

In some situations the *value* of an expression will vary depending on the types of the arguments.

# Strong typing

---

In CS, a system or language is *strongly typed* if variables and values all have a specified data type, and operations require their arguments to have the correct data types.

```
>>> "2"+3  
  
Traceback (most recent call last):  
  File "<pyshell#1>", line 1, in <module>  
    "2"+3  
TypeError: cannot concatenate 'str' and 'int' objects  
>>> "2"+"3"  
'23'  
>>> 2+3  
5  
>>> |
```

This snippet of Python shows that "2"+3 fails because "2" is a string and 3 is an integer, and Python does not know to "add" strings to integers.

# Weak typing

---

A system or language is *weakly typed* if the system is more permissive about combining variables and/or values of different types.

A weakly-typed system will try to *find a way* to complete an operation even if the types of the arguments don't really match.

```
sqlite> SELECT "2" + 3;  
5  
sqlite>
```

SQLite is weakly typed — it decides to treat the string as a number, and then do the addition operation.

## More examples

---

```
|sqlite> SELECT "2" + "3";  
|5  
|sqlite> SELECT "2" + "a";  
|2
```

```
sqlite> SELECT 2 + a;  
Error: no such column: a
```

## Safety or flexibility

---

Should the system *protect* the user or *trust* the user?

- ▶ Strong typing is *safer*

*If the user is asking for the sum of a string and an integer, then that is a logical error and so it should be prohibited.*

- ▶ Weak typing is *more flexible*

*Sometimes integers are represented by strings, so if the user asks for the sum of a string and an integer, the system should try to comply.*

(Personally, I like strongly-typed languages, but I have clever friends and colleagues whose opinions I respect who prefer weakly-typed languages.)

# Types in SQLite

---

From a user perspective, SQLite has just five datatypes<sup>1</sup>:

- ▶ **NULL**  
The value is **NULL**
- ▶ **INTEGER**  
The value is an *integer*, such as 1 or -23.
- ▶ **REAL**  
The value is *floating point number*, such as 1.2 or -3.14.
- ▶ **TEXT**  
The value is a *text string*, such as "Relational Databases"
- ▶ **BLOB**  
The value is a **Binary Large OBject** – basically just a long sequence of bytes with no particular interpretation.

---

<sup>1</sup>for slightly complicated reasons, this is not the complete truth

# SQLite

---

In fact, SQLite essentially ignores the type declarations.

If you declare a table using other types (say from MySQL)<sup>2</sup>,  
SQLite will not complain.

SQLite only assigns a type to a *value*, not a column, and tries to make sense of expressions using seemingly-incompatible types.

For this unit, *use only* the five SQLite types in your code.

---

<sup>2</sup>or even just random words!

# Booleans in SQL

---

SQL uses many *boolean expressions*, most obviously after `WHERE`.

```
WHERE homeScore = 100 ...
```

```
WHERE homeScore > awayScore ...
```

```
WHERE ABS(homeScore - awayScore) <= 10 ...
```

```
WHERE (homeTeam = "Sydney") OR (awayTeam = "Sydney") ...
```

## Booleans in SQLite

---

In SQLite, boolean expressions evaluate to 1 (if true) or 0 (if false).

```
| sqlite> SELECT 1+2 == 3;  
| 1  
| sqlite> SELECT 1+2 > 3;  
| 0  
| sqlite> |
```

The values `TRUE` and `FALSE` are *aliases* for 1 and 0 respectively.

You might use these to enhance the *readability* of your SQLite code, or for *compatibility* with other SQL implementations.

## Non-zeros are true

---

If you use a *numeric* or *text* value in a *boolean context*, then:

- ▶ Any non-zero number (not just 1) is treated as *true*
- ▶ The value 0 is treated as *false*
- ▶ Any text value is treated as *false*

What on earth does this do?

```
SELECT 1+"true" WHERE true+-1;
```

# CITS1402

## Relational Database Management Systems

### Video 11 — Operators

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Operators

---

In SQL it is often necessary to do *calculations* or *comparisons* based on the values of one or more columns.

```
SELECT *, (homeScore + awayScore) AS totalScore  
FROM AFLResult  
ORDER BY totalScore DESC  
LIMIT 10;
```

The calculation uses the *binary arithmetic operator* +

- ▶ *Binary* means that it takes two arguments (inputs)
- ▶ *Arithmetic* means that arguments and output are numbers

# Arithmetic Operators I

---

Most of the arithmetic operators are *binary operators* that obey the usual mathematical rules of precedence:

- ▶ Multiplication \*, division / and integer remainder %

```
sqlite> SELECT 3 * 4;  
12  
sqlite> SELECT 3 * 1.5;  
4.5  
sqlite> SELECT 6 / 4;  
1  
sqlite> SELECT 6 / 4.0;  
1.5  
sqlite> SELECT 6 % 4;  
2  
sqlite> 
```

As in C (the programming language), if both arguments are *integers*, so is the result.

## Arithmetic Operators II

---

As usual, addition and subtraction have lower priority than multiplication and division.

- ▶ Addition +, Subtraction -

```
sqlite> SELECT 2 + 3;  
5  
sqlite> SELECT 2 + 3 * 6;  
20  
sqlite> SELECT (2 + 3) * 6;  
30  
sqlite> SELECT 2 - 3 * 4 - 3;  
-13  
sqlite>
```

As usual, *brackets* can be used to override the standard precedence.

# Arithmetic Operators III

---

These are *bitwise operators* as found in C.

- ▶ Left shift <<, right shift >>, bitwise AND &, bitwise OR |

```
sqlite> SELECT 21 << 2;  
84  
sqlite> SELECT 21 >> 2;  
5  
sqlite> SELECT 21 & 7;  
5  
sqlite> SELECT 21 | 7;  
23  
sqlite> 
```

You *do not need* to learn bitwise operators for CITS1402.

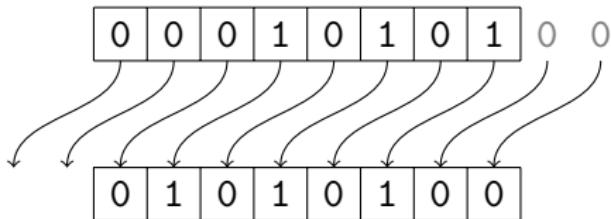
# Bitwise operators

---

Numbers are stored as bit-patterns

0	0	0	1	0	1	0	1
128	64	32	16	8	4	2	1

- ▶ Shift left by 2 bits



## Comparison operators

---

Comparison operators *compare values* and return *boolean values* indicating the result of the comparison — most are binary operators.

- ▶ Less than < and less than or equal to <=
- ▶ More than > and more than or equal to >=

```
sqlite> SELECT 2 < 3;
1
sqlite> SELECT 3 < 3;
0
sqlite> SELECT 3 <= 3;
1
sqlite>
```

# Comparison operators for strings

---

Strings are compared *lexicographically* (i.e. alphabetically).

```
sqlite> SELECT "Dog" < "Cat";
0
sqlite> SELECT "Dog" < "Dogs";
1
sqlite> SELECT "Cat" < "Camel";
0
sqlite> 
```

What about non-letters?

# ASCII

---

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	'	112	p		
33	!	49	1	65	A	81	Q	97	a	113	q		
34	"	50	2	66	B	82	R	98	b	114	r		
35	#	51	3	67	C	83	S	99	c	115	s		
36	\$	52	4	68	D	84	T	100	d	116	t		
37	%	53	5	69	E	85	U	101	e	117	u		
38	&	54	6	70	F	86	V	102	f	118	v		
39	'	55	7	71	G	87	W	103	g	119	w		
40	(	56	8	72	H	88	X	104	h	120	x		
41	)	57	9	73	I	89	Y	105	i	121	y		
42	*	58	:	74	J	90	Z	106	j	122	z		
43	+	59	.	75	K	91	[	107	k	123	{		
44	,	60	<	76	L	92	\	108	l	124			
45	-	61	=	77	M	93	]	109	m	125	}		
46	.	62	>	78	N	94	^	110	n	126	~		
47	/	63	?	79	O	95	_	111	o	127	[backspace]		

# Equality and Inequality

---

- ▶ Two choices for equality: `=` and `==`
- ▶ Two choices for inequality: `!=` and `<>`

```
sqlite> SELECT 2*2 == 4;  
1  
sqlite> SELECT 2 + 2 == 4;  
1  
sqlite> SELECT 2 * 2 = 4;  
1  
sqlite> SELECT 2 + 3 != 4;  
1  
sqlite> SELECT 2 + 3 <> 5;  
0  
sqlite> 
```

## Comparisons with `NULL`

---

Remember that a value of `NULL` means “not yet known, undefined or inapplicable”.

Students who have passed:

```
SELECT sNum  
FROM Enrolment  
WHERE mark >= 50;
```

Students who have failed:

```
SELECT sNum  
FROM Enrolment  
WHERE mark < 50;
```

What happens if a row has `NULL` for `mark`?

## Separate operators for `NULL`

---

Sometimes you need to check if the column is *actually storing* the value `NULL`, but these two expressions

`mark == NULL`      `mark <> NULL`

have no value *no matter what* value is in the column `mark`.

Operators `IS` and `IS NOT` test for the literal value `NULL`.

```
SELECT sNum  
FROM Enrolment  
WHERE mark IS NULL;
```

## BETWEEN

---

One of the few *ternary* operators (because it takes 3 arguments)

- ▶ ... BETWEEN ... AND ...

```
sqlite> SELECT *
...>   FROM AFLResult
...> WHERE homeScore - awayScore BETWEEN -10 AND 10;
2012|1|Fremantle|Geelong|105|101
2012|1|North Melbourne|Essendon|102|104
2012|1|Port Adelaide|St Kilda|89|85
2012|2|Geelong|Hawthorn|92|90
2012|4|West Coast|Hawthorn|51|46
2012|4|Geelong|Richmond|75|65
2012|5|Collingwood|Essendon|80|79
```

## To come later

---

- ▶ **IN and NOT IN**

These test if a value is present or is not present in a *list* —  
but we have not seen how to create lists yet.

- ▶ **LIKE**

This is used for *pattern matching* in strings, which is hugely important in practice.

CITS1402  
Relational Database Management Systems

Video 12 — Relational Algebra

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Relational Algebra

---

The *mathematical theory* underlying relational databases.

Traditional component of courses on relational databases.

Useful for theoretical discussion of relational databases, but not really important for practical use of relational databases.

We'll just give a very brief overview here.

## Expressions in Algebra

---

An expression in (normal) *algebra* is something like

$$(x + y)(x - y)$$

where  $x$  and  $y$  are *variables* combined in a particular way with arithmetic operators.

If we “plug in” specific *numbers* for  $x$  and  $y$ , then the value of the expression can be calculated, and it is itself a *number*.

# Algebraic Manipulation

---

From the *rules of algebra*, we know that

$$x^2 - y^2 = (x - y)(x + y)$$

and so we know that

$x^2 - y^2$  can always be replaced with  $(x - y)(x + y)$ .

In this sort of way, algebraic expressions can be *rewritten* in a way that is guaranteed to be *mathematically correct*.

Why? One might be *cheaper* or *safer* to calculate.

# How much faster is this really?

---

This is SageMath code that just calculates  $x^2 - y^2$  for 1000000 pairs of random numbers.

```
: 1 %%time
: 2 vals=[]
: 3 for it in range(10000000):
: 4     x = random()
: 5     y = random()
: 6     vals.append(x^2 - y^2)|
```

CPU times: user 12 s, sys: 279 ms, total: 12.3 s  
Wall time: 12.3 s

```
: 1 %%time
: 2 vals=[]
: 3 for it in range(10000000):
: 4     x = random()
: 5     y = random()
: 6     vals.append((x+y)*(x-y))|
```

CPU times: user 7.88 s, sys: 281 ms, total: 8.16 s  
Wall time: 8.2 s

The speed gained by this one change is astonishing.

# Relations

---

Mathematically, a *relation* is just a *set of tuples* — consider the following table called  $R$

A	B	C
1	0	1
2	1	1
1	3	0

It has three rows so

$$R = \{(1, 0, 1), (2, 1, 1), (1, 3, 0)\}$$

# Relational Algebra

---

A legal expression in *relational algebra* is built from

- ▶ Variables representing relations
- ▶ Set-theoretic operators
- ▶ “Relational” operators

composed according to the rules of relational algebra.

Every *legal expression* in relational algebra determines a *relation*.

# Expressions in Relational Algebra

---

Here is a simple *expression* using the set theory operator  $\cup$ .

$$R \cup S$$

and a simple *identity*

$$R \cup S = S \cup R.$$

The operator  $\cup$  is the *union* operator which creates a set containing everything that is in either  $R$  or  $S$ .

# Is this in SQL?

---

```
4 CREATE TABLE R(A INT, B INT, C INT);
5 CREATE TABLE S(A INT, B INT, C INT);
6
7 INSERT INTO R VALUES(1,0,1);
8 INSERT INTO R VALUES(2,1,1);
9 INSERT INTO R VALUES(1,3,0);
10
11 INSERT INTO S VALUES(2,1,1);
12 INSERT INTO S VALUES(2,2,2);
13 INSERT INTO S VALUES(3,0,0);
14
15 SELECT * FROM R
16 UNION
17 SELECT * FROM S;
```

Grid



Total rows loaded: 5

	A	B	C
1	1	0	1
2	1	3	0
3	2	1	1
4	2	2	2
5	3	0	0

# Expressions in Relational Algebra

---

Here is an *expression* in relational algebra using the operator  $\cap$

$$R \cap S$$

and a simple *identity*

$$R \cap S = S \cap R.$$

The operator  $\cap$  is the *intersection* operator which creates a set containing every tuple that is in *both*  $R$  and  $S$ .

# Is this in SQL?

---

```
19 SELECT * FROM R  
20 INTERSECT  
21 SELECT * FROM S;
```

The screenshot shows a database management system interface. At the top, there is a code editor window containing the provided SQL query. Below the code editor is a toolbar with various icons for database operations. Underneath the toolbar is a results grid displaying the output of the query. The results grid has three columns labeled A, B, and C, and one row of data with values 1, 2, and 1 respectively.

A	B	C
1	2	1

## Who needs to know it?

---

Who *really needs* to know relational algebra?

- ▶ Most database *users* can get by without it.
- ▶ Database *developers* may find it useful for query development.
- ▶ Database *implementors* need it for query optimization.

You should know that it exists and be able to recognise it.

# Relational Operators

---

The two main *relational operators* are

- ▶ The *projection operator*  $\pi$  (the Greek letter “pi”)  
A relation is projected onto a subset of its columns
- ▶ The *selection operator*  $\sigma$  (the Greek letter “sigma”)  
A subset of the rows is chosen based on a boolean condition

## Projection

---

The *projection operator* is denoted  $\pi$  (the Greek letter “pi”).

Suppose that  $R$  is a relation and that  $C$  is a *list of column names*.

Then the relation

$$\pi_C(R)$$

is obtained from  $R$  by taking only the columns listed in  $C$  from each row.

Any duplicate columns are removed from the resulting table.

## Example Relation

---

Suppose  $R$  is the following relation

customerId	name	address	accountMgr
1121	Bunnings	Subiaco	137
1122	Bunnings	Claremont	137
1211	Mitre 10	Myaree	186
1244	Mitre 10	Joondalup	186
1345	Joe's Hardware	Nedlands	204
1399	NailsRUs	Jolimont	361

## Example Projections

---

Then  $\pi_{\text{customerID}, \text{name}}(R)$  is

customerID	name
1121	Bunnings
1122	Bunnings
1211	Mitre 10
1244	Mitre 10
1345	Joe's Hardware
1399	NailsRUs

We have *projected* the relation onto the two named columns, thus obtain a relation with fewer columns.

## Another example projection

---

Now  $\pi_{\text{name}}(R)$  is the relation

name
Bunnings
Mitre 10
Joe's Hardware
NailsRUs

A relation is a *set* so duplicate rows are removed.

## In SQL

---

It should be clear that there is a direct relationship between

- ▶ the *projection operator*  $\pi$ , and
- ▶ the `SELECT` columns `FROM` statement

But SQLite *does not* automatically remove duplicate rows (for efficiency).

## Selection

---

Projection is an operation that extracts *columns* from a relation, while *selection* is the operation that extracts *rows* from a relation.

The selection operator is denoted by  $\sigma$  (Greek letter “sigma”).

If  $R$  is a relation and  $B$  is a boolean expression, then

$$\sigma_B(R)$$

is the relation containing only the rows for which  $B$  is true.

## Example Selection

---

If  $R$  is the relation defined above, then

$$\sigma_{\text{customerID} < 1300}(R)$$

is the relation

customerId	name	address	accountMgr
1121	Bunnings	Subiaco	137
1122	Bunnings	Claremont	137
1211	Mitre 10	Myaree	186
1244	Mitre 10	Joondalup	186

## Complex Expressions

---

Complex expressions can be built up by *composing* operators.

$$\pi_{\text{accountMgr}}(\sigma_{\text{name}='Bunnings'}(R))$$

- ▶ Choose rows that have name Bunnings (keep all columns)
- ▶ From this smaller table keep only the accountMgr column

## In SQL

---

The condition in an expression involving  $\sigma$  is directly analogous to the [WHERE](#) statement of SQL.

```
SELECT accountMgr  
FROM R  
WHERE name = 'Bunnings';
```

## Complex boolean expressions

---

The boolean functions that can be used as the selection condition are combinations using  $\wedge$  (for AND) and  $\vee$  (for OR) of *terms* of the form

*attribute op constant*

or

*attribute1 op attribute2*

where *op* is a comparison operator in the set

$\{<, \leqslant, =, \neq, \geqslant, >\}$ .

# CITS1402

## Relational Database Management Systems

### Video 13 — Summary Functions

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Summarising Data

---

One of the main uses of a database is to *summarize* the data it contains, in particular to provide *statistical data*.

The main summary / aggregate functions are

- ▶ COUNT – to *count* rows
- ▶ SUM – to *add up* the values in a column
- ▶ MIN – to find the *minimum* value in a column
- ▶ MAX – to find the *maximum* value in a column
- ▶ AVG – to find the *average value* in a column
- ▶ STD – to find the *standard deviation* of the values in a column

# The key point

---

The *critical point* to remember is the following:

When using a summary / aggregate function, a *group of rows* is replaced by a *single* row containing “*summary information*” about that group of rows.

In the simplest case, the group of rows is just the entire table.

The image shows a database interface with two panes. The left pane displays the SQL query:

```
1 SELECT *
2 FROM R;
```

The right pane shows the result of the query:

	AVG(A)
1	1.3333333333333333

The interface includes standard database navigation buttons (refresh, save, etc.) and a toolbar with icons for search, insert, update, delete, and other operations.

# Let's count!

---

The most heavily used summary function is undoubtedly [COUNT](#).

A screenshot of a database query interface. At the top, there are tabs for "Query" and "History", with "Query" being active. Below the tabs is a code editor containing the following SQL query:

```
1 SELECT COUNT(*)
2 FROM Track;
```

Below the code editor are two buttons: "Grid view" (which is selected) and "Form view". Underneath these buttons is a toolbar with several icons. To the right of the toolbar, it says "Total rows loaded: 1". The main results area shows a single row of data:

COUNT(*)
1 3503

Notice the rather *strange syntax*.

# Tennis Database

---

Each year in January, the Australian Open Tennis tournament is held in Melbourne.

The two main events are the Men's Singles and Women's Singles, which are *knockout tournaments* that start with 128 players each.

So in Round 1 (R1) there are 64 matches, with the 64 winners going to Round 2 (R2) and the 64 losers going home.

Then there are 32 matches in R2, 16 matches in R3, 8 matches in R4, 4 matches in R5 (quarter-finals), 2 matches in R6 (semi-finals) and 1 match in R7 (the final).

# The AustOpen database

---

I have extracted tennis match results collated by Jeff Sackmann<sup>1</sup> to make two tables called **ATPResult** and **WTAResult**.

```
sqlite> .schema WTAResult
CREATE TABLE WTAResult (
    yr INTEGER,
    winnerName TEXT,
    winnerHand TEXT,
    winnerHeight INTEGER,
    winnerCountry TEXT,
    winnerAge REAL,
    loserName TEXT,
    loserHand TEXT,
    loserHeight INTEGER,
    loserCountry TEXT,
    loserAge REAL,
    score TEXT,
    minutes INTEGER);
```

ATP = Association of Tennis Professionals

WTA = Women's Tennis Association

---

<sup>1</sup>Used with permission: [https://github.com/JeffSackmann/tennis\\_atp](https://github.com/JeffSackmann/tennis_atp)

## Not a tennis fan?

---

Sports are convenient for learning databases because sports data tends to be *comprehensible*, *clean* and *complete*.

No tennis-specific knowledge is required.

The only flaws in these two tables arise from *missing data* - for example, not every player has a value for *height*.

If any actual value is unknown, then the value stored in the database is *NULL*.

# Long tennis matches

---

How long was the *longest* Men's Singles Match from 2000–2022?

```
SELECT MAX(minutes)  
FROM ATPResult;
```

```
SELECT MAX(minutes)  
FROM WTAResult;
```

```
sqlite> SELECT MAX(minutes) FROM WTAResult;  
284  
sqlite> SELECT MAX(minutes) FROM ATPResult;  
353  
sqlite> 
```

# Which were these matches

---

In regular SQL, to find the *actual match* that lasted that long, we need to run a *second query*.

The screenshot shows a MySQL Workbench interface. At the top, there are tabs for 'Query' and 'History'. Below the tabs is a code editor containing the following SQL query:

```
1 SELECT yr, winnerName, loserName, score
2 FROM WTAResult
3 WHERE minutes = 284;
```

Below the code editor is a results grid. The grid has a header row with columns: yr, winnerName, loserName, and score. The data row below shows the result of the query:

	yr	winnerName	loserName	score
1	2011	Francesca Schiavone	Svetlana Kuznetsova	6-4 1-6 16-14

The results grid includes navigation buttons (first, previous, next, last) and a status message: 'Total rows loaded: 1'. There are also 'Grid view' and 'Form view' buttons at the top of the results area.

# Average matches

---

The *average length* is easy to find

The screenshot shows a database query interface. At the top, there is a blue "Query" button. Below it, the SQL code is displayed:

```
1 SELECT AVG(minutes)
2 FROM ATPResult;
3
```

Below the code, there is a "Grid view" section with various icons for filtering and sorting. The status bar indicates "Total rows loaded: 1". A single row of data is shown:

AVG(minutes)
148.22889498970488

But this is the *overall average* for 23 years worth of matches.

# Average match length in 2022?

---

```
SELECT AVG(minutes)
FROM ATPResult
WHERE yr = 2022;
```

```
1 SELECT AVG(minutes)
2 FROM ATPResult
3 WHERE yr = 2022;
4
```

Total rows loaded: 1

	AVG(minutes)
1	163.25984251968504

# How does this work?

---

- ▶ The `FROM ATPResult` starts with the whole table
- ▶ The `WHERE` keeps only the rows from 2022
- ▶ The `AVG(minutes)` says to take the average of minutes

Query History

```
1 SELECT yr, winnerName, loserName, minutes
2 FROM ATPResult
3 WHERE yr = 2022;|
```

Total rows loaded: 127

Grid view Form view

	yr	winnerName	loserName	minutes
1	2022	Miomir Kecmanovic	Salvatore Caruso	116
2	2022	Tommy Paul	Mikhail Kukushkin	99
3	2022	Oscar Otte	Chun Hsin Tseng	110
4	2022	Lorenzo Sonego	Sam Querrey	134
5	2022	Gael Monfils	Federico Coria	95

# Has the average changed over time?

---

yr	yrAvg
2000	136
2001	141
2002	146
2003	139
2004	134
2005	140
2006	149
2007	142
2008	146
2009	158
2010	151
2011	146
2012	157
2013	151
2014	149
2015	147
2016	149
2017	149
2018	155
2019	156
2020	154
2021	147
2022	163

How has the *average match length* changed over the years?

We could run 23 queries, with each query changing the year, but ideally we'd like to use *one query* to produce a table with a row *for each year*.

Later we'll see how to form the rows into *groups* and apply a summary function to *each group individually*.

CITS1402  
Relational Database Management Systems

Video 14 — GROUP BY

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Querying

---

The basic **SELECT** statement that we are using is:

```
SELECT <column_names>
FROM <table_references>
WHERE <row_conditions>
GROUP BY <group_columns>
HAVING <group_conditions>
ORDER BY <sorting_columns>
LIMIT <number_rows>
```

# Grouping

---

The `GROUP BY` statement is one of *the most important* we've seen.

This statement *groups together* rows and then applies the summary/aggregate functions *to each group* separately.

There is *one output row* for each *group*.

# Summary for every year

---

```
SELECT yr, AVG(minutes)
FROM ATPResult
GROUP BY yr;
```

The screenshot shows a MySQL Workbench interface titled "AustOpen". The SQL editor contains the following query:

```
1 SELECT yr, AVG(minutes)
2 FROM ATPResult
3 GROUP BY yr;
```

The results grid displays the average minutes for each year from 2000 to 2013. The table has two columns: "yr" and "AVG(minutes)". The data is as follows:

	yr	AVG(minutes)
1	2000	136.4488188976378
2	2001	141.1968503937008
3	2002	146.4488188976378
4	2003	139.32539682539684
5	2004	134.45669291338584
6	2005	140.4724409448819
7	2006	149.23809523809524
8	2007	142.17322834645668
9	2008	145.74015748031496
10	2009	157.736
11	2010	151.3015873015873
12	2011	146.21259842519686
13	2012	157.1031746031746
14	2013	150.9448818897638

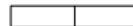
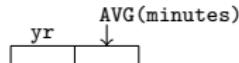
Total rows loaded: 23

# Code Dissection

---

Rows are divided into *groups* according to the value of *yr*.

yr			minutes

Then `SELECT yr, AVG(minutes)` is performed on *each group separately*.

# The winningest countries

---

```
SELECT yr, winnerCountry,  
       COUNT(*)  
  FROM WTAResult  
 GROUP BY yr, winnerCountry;
```

The GROUP BY names *two columns.*

```
1 SELECT yr, winnerCountry, COUNT(*)  
2 FROM WTAResult  
3 GROUP BY yr, winnerCountry;
```

	yr	winnerC	COUN
1	2000	ARG	2
2	2000	AUS	5
3	2000	AUT	4
4	2000	BEL	6
5	2000	BLR	2
6	2000	CAN	3
7	2000	CHN	2
8	2000	COL	1
9	2000	CRO	2
10	2000	CZE	4
11	2000	ESP	11
12	2000	FRA	14
13	2000	GBR	1
14	2000	GER	4

## More than one GROUP BY column

---

Rows in a group have same value for *both yr and winnerCountry*.

The screenshot shows a SQL editor window titled "SQL editor 1". The query tab contains the following SQL code:

```
1 SELECT *
2 FROM WTAResult
3 WHERE yr = 2000 AND winnerCountry = 'AUS';
```

The results are displayed in a grid view, showing 5 rows of data. The columns are labeled: yr, winnerName, winnerHi, winnerHr, winnerC, winnerRt, loserName. The data is as follows:

	yr	winnerName	winnerHi	winnerHr	winnerC	winnerRt	loserName
1	2000	Alicia Molik	R	182	AUS	18.9	Silvija Talaja
2	2000	Bryanne Stewart	R	174	AUS	20.1	Maria Vento Kabchi
3	2000	Nicole Pratt	R	163	AUS	26.8	Maria Sanchez Lorenzo
4	2000	Alicia Molik	R	182	AUS	18.9	Karina Habsudova
5	2000	Bryanne Stewart	R	174	AUS	20.1	Emmanuelle Gagliardi

The second group had **yr = 2000** and **winnerCountry = 'AUS'**.

## Forming the summary row

---

The *summary row* is formed according to the `SELECT` statement.

```
SELECT yr, winnerCountry, COUNT(*)  
FROM ...
```

	yr	winnerName	winnerHi	winnerHt	winnerC	winnerAç	loserName
1	2000	Alicia Molik	R		182 AUS	18.9	Silvija Talaja
2	2000	Bryanne Stewart	R		174 AUS	20.1	Maria Vento Kabchi
3	2000	Nicole Pratt	R		163 AUS	26.8	Maria Sanchez Lorenzo
4	2000	Alicia Molik	R		182 AUS	18.9	Karina Habsudova
5	2000	Bryanne Stewart	R		174 AUS	20.1	Emmanuelle Gagliardi

Every column named in `SELECT` is either

- ▶ A `GROUP BY` column, or
- ▶ An aggregate function

## Bare Columns

---

A *bare column* is a column that is

- ▶ Not a summary or aggregate function, and
- ▶ Not in GROUP BY.

```
SELECT yr, winnerCountry, winnerHeight, COUNT(*)  
FROM WTAresult  
GROUP BY yr, winnerCountry;
```

## Dealing with a bare column

---

	yr	winnerName	winnerHt	winnerHt	winnerC	winnerAç	loserName
1	2000	Alicia Molik	R	182	AUS	18.9	Silvija Talaja
2	2000	Bryanne Stewart	R	174	AUS	20.1	Maria Vento Kabchi
3	2000	Nicole Pratt	R	163	AUS	26.8	Maria Sanchez Lorenzo
4	2000	Alicia Molik	R	182	AUS	18.9	Karina Habsudova
5	2000	Bryanne Stewart	R	174	AUS	20.1	Emmanuelle Gagliardi

The problem is that a bare column might take *different values* within a single group.

What value should SQL take for `winnerHeight` ?

## More examples

---

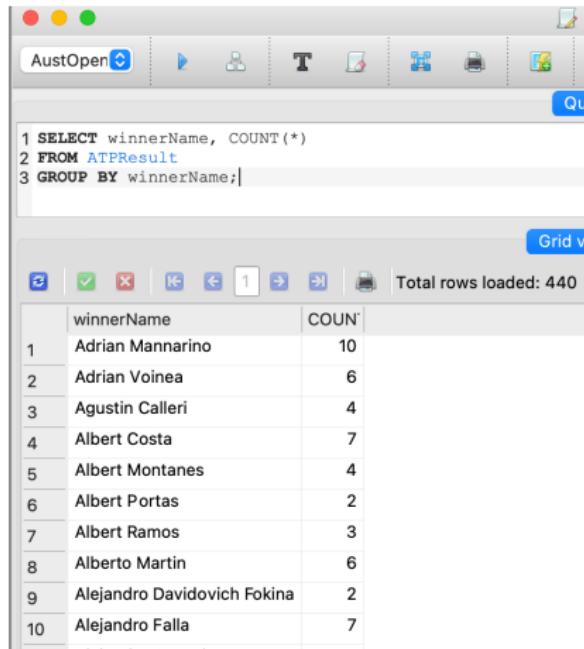
How many total wins has each male player had?

- ▶ Group rows according to the winner's name, and
- ▶ Count the rows in each group.

```
SELECT winnerName, COUNT(*)
FROM ATPResult
GROUP BY winnerName;
```

# Matches won

---



The screenshot shows a MySQL Workbench interface with a query editor and a results grid.

**Query Editor:**

```
1 SELECT winnerName, COUNT(*)
2 FROM ATPResult
3 GROUP BY winnerName;
```

**Results Grid:**

	winnerName	COUNT(*)
1	Adrian Mannarino	10
2	Adrian Voinea	6
3	Agustin Calleri	4
4	Albert Costa	7
5	Albert Montanes	4
6	Albert Portas	2
7	Albert Ramos	3
8	Alberto Martin	6
9	Alejandro Davidovich Fokina	2
10	Alejandro Falla	7

Total rows loaded: 440

## Column aliases

---

By default, the *column names* of the output table is whatever was in `SELECT`.

The `AS` statement is used to create a sensible *column alias*.

```
SELECT winnerName, COUNT(*) AS numWins  
FROM ATPResult  
GROUP BY winnerName;
```

## Using the alias

---

The alias can then be used, for example in a `ORDER BY` statement, to give the output in a more suitable order.

```
SELECT winnerName, COUNT(*) AS numWins  
FROM ATPResult  
GROUP BY winnerName  
ORDER BY numWins DESC;
```

# The best ever

---

```
1 SELECT winnerName, COUNT(*) AS numWins
2 FROM ATPResult
3 GROUP BY winnerName
4 ORDER BY numWins DESC;
5 |
```



Total rows lo

	winnerName	numWins
1	Roger Federer	103
2	Novak Djokovic	82
3	Rafael Nadal	76
4	Andy Murray	49
5	Tomas Berdych	47
	Stan Wawrinka	44

## The losers

---

So far the code only counts players who have *won at least one match*.

If someone plays, but immediately loses, then their name will only appear in the `loserName` column.

But perhaps we want to include them in the list, even if they have 0 in the `numWins` column?

This requires rather more complex code.

CITS1402  
Relational Database Management Systems

Video 15 — Entity Relationship Diagrams I

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Database Design Process

---

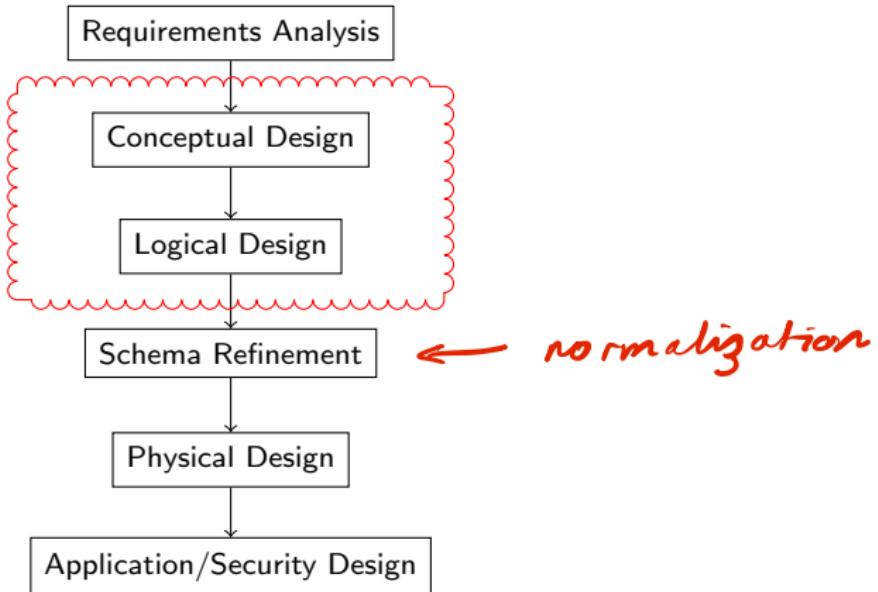
Ramakrishnan & Gehrke identify six steps in *designing* a database

- ▶ Requirements Analysis
- ▶ Conceptual Design
- ▶ Logical Design
- ▶ Schema Refinement
- ▶ Physical Design
- ▶ Application & Security Design

# Database Design

---

ER-Modelling



# Requirements Analysis

---

*Requirements Analysis* is the process of determining *what* the database is to be used for.

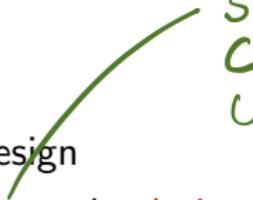
User groups interviewed to determine

- ▶ the desired functionality,
- ▶ the types of data stored,
- ▶ what queries are likely to made.

Non-technical discussions that explain the “business logic” to the developers.

# Conceptual & Logical Design

---

- 
- Student  
Course  
Unit -
- “what  
things  
are like”
- ✓ how to  
do this
- ▶ Conceptual Design
    - Identify **entities** and **relationships** and construct an **entity-relationship diagram** (ERD).
  - ▶ Logical Design
    - Devise the **database schema** (that is, the tables and the names/types of their columns) based on the structure revealed in the ERD.

## After the modelling

---

More steps, but mostly outside the scope of this unit

- ▶ Mathematical analysis and refinement of the schema
- ▶ Performance-based decisions on indexing, machine capacities, required performance etc.
- ▶ Interfacing with client applications and security

## Iterative Process

---

As with all software engineering processes, implementation of a later stage often reveals:

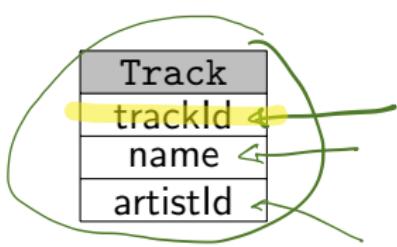
- ▶ New or inadequately-analysed use cases
- ▶ Additional or enhanced functionality

Usually there are several iterations through the process are needed before settling on a final design.

# Diagramming

---

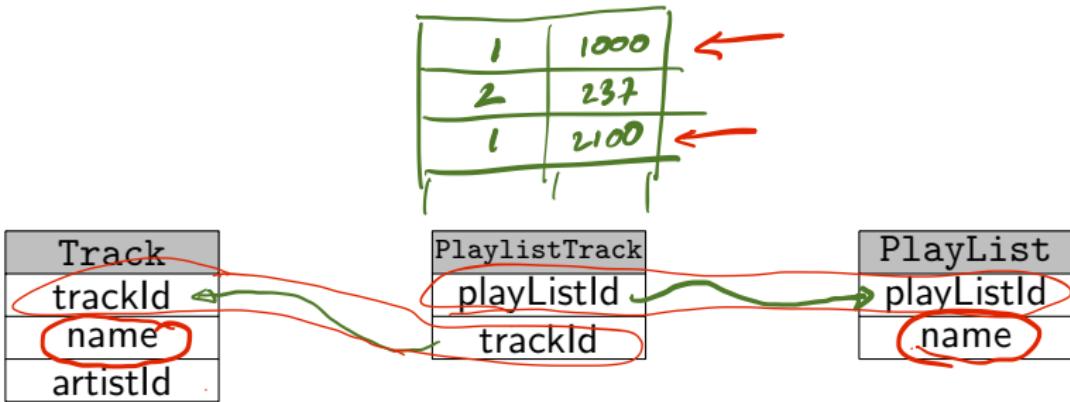
When *designing* a new database or *understanding* an existing database it is often convenient to *draw pictures*.



The pictures show the *general structure* of the database, not the current values.

# Diagramming

---



Here, **PlaylistTrack** is called a **bridge table**.

## The bridge table

---

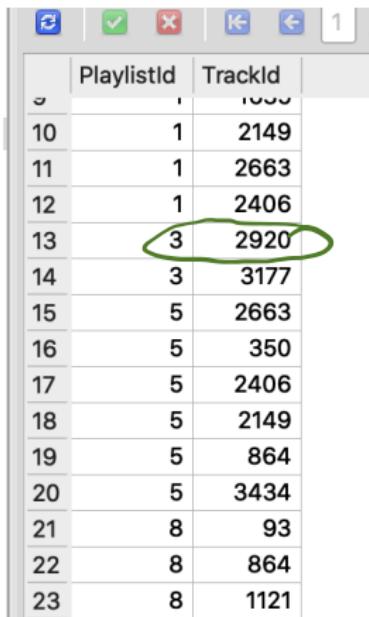
A **Track** can be on many **PlayList**s and a **PlayList** can contain many **Tracks**.

This is called a ***many-to-many*** relationship between **Track** and **Playlist**.

A bridge table is the **design pattern** used to implement many-to-many relationships.

## In Chinook

---



A screenshot of a database application showing a grid of data. The grid has a header row with columns labeled 'PlaylistId' and 'TrackId'. Below the header are 14 data rows, each containing two numerical values. A green oval highlights the third row, which contains the values '3' and '2920'. The application interface includes a toolbar at the top with icons for refresh, save, delete, and back/forward navigation.

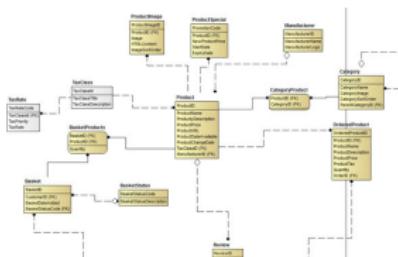
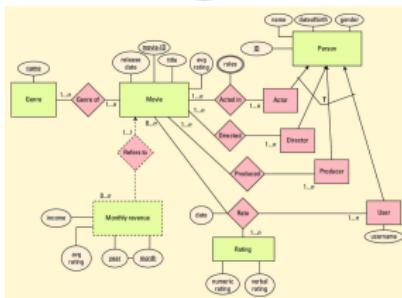
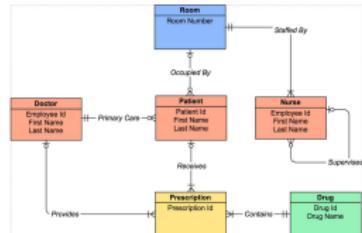
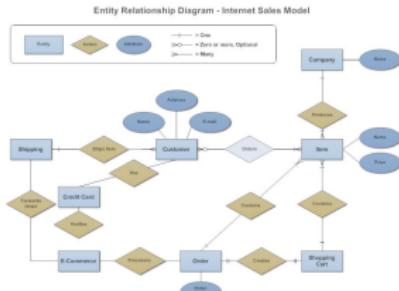
	PlaylistId	TrackId
10	1	2149
11	1	2663
12	1	2406
13	3	2920
14	3	3177
15	5	2663
16	5	350
17	5	2406
18	5	2149
19	5	864
20	5	3434
21	8	93
22	8	864
23	8	1121

Each row represents a *particular track* occurring on a *particular playlist*.

e.g. Track 2920 is on Playlist 3.

# Diagramming

Numerous ways to diagram a *database schema*.



# Entity Relationship Diagrams

---

We will focus on *just one* of the many diagramming conventions.

Normally any correct way of expressing a SQL query is fine.

~~Diagramming is an exception~~—the people marking labs/projects cannot be familiar with all of the dozens of diagramming methods.

So just for this one part of the unit, you have stick to *one* *particular way* of diagramming.

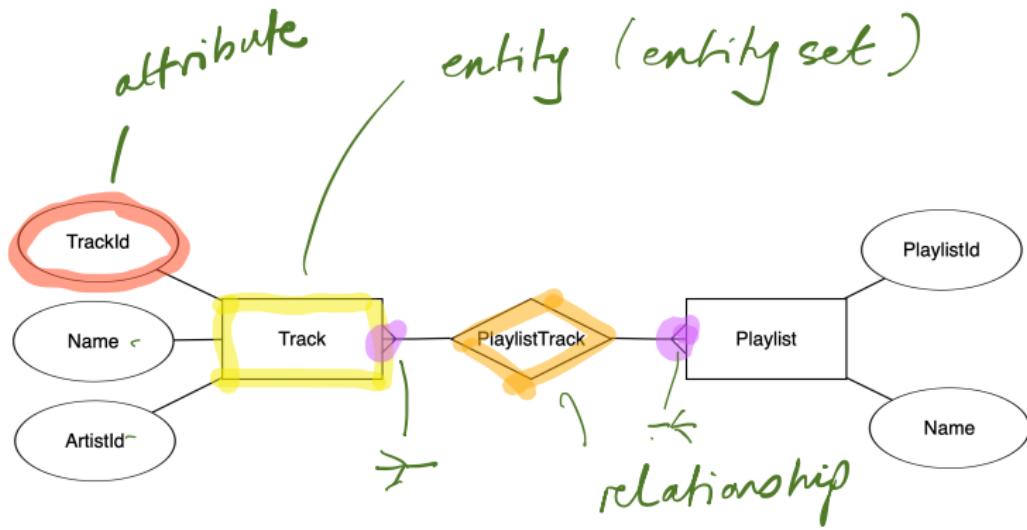
The website [erdplus.com](http://erdplus.com) provides a free online tool that allows you to create a *particular style* of Entity-Relationship Diagram.

Using this tool for labs and projects will ensure that your diagrams will meet the basic requirements.

ERDPlus provides straightforward “classical” *entity-relationship diagrams* as they were first introduced in the 1970s.

# ERD

---



CITS1402  
Relational Database Management Systems

Video 16 — HAVING

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Querying

---

The basic **SELECT** statement that we are using is:

```
SELECT <column_names>
FROM <table_references>
WHERE <row_conditions>
GROUP BY <group_columns>
HAVING <group_conditions>
ORDER BY <sorting_columns>
LIMIT <number_rows>
```

## Groups

---

Recall the “row-processing” that occurs when summary functions are used.

Starting with the table created by `FROM`,

- ▶ The `GROUP BY` clause divides the rows into groups
- ▶ The *aggregate* or *summary* functions are applied to each group
- ▶ The output table has one *summary row* per group

But what if you only want output for *some groups*?

## The HAVING statement

---

The **HAVING** statement specifies *extra conditions* that are applied *after* the summary rows are formed and *before* they are output.

Only summary rows that *satisfy the conditions* are output

Note: the **WHERE** conditions choose rows *before grouping* and the **HAVING** conditions choose (summary) rows *after grouping*.

## Frequent winners

---

Which players have won at least 40 matches in the Australian Open database.

Each row of `ATPResult` or `WTAResult` contains the name of one match-winner.

- ▶ Need to *form groups* according to player's *name*.
- ▶ Need to *count* the rows in each group
- ▶ Need to *keep* only those with at least 40 matches

## Frequent winners

---

```
SELECT winnerName, COUNT(*)  
FROM WTAResult  
GROUP BY winnerName  
HAVING COUNT(*) >= 40;
```

# Code dissection |

---

- ▶ The **FROM** statement says to use all rows
- ▶ The **GROUP BY** statement forms groups on winner names

Year	Winner	Country	Opponent	Country	Score
2005	Abigail Spears	USA	Meghann Shaughnessy	USA	1-6 6-2 6-2
2005	Abigail Spears	USA	Tatiana Golovin	FRA	7-5 6-1
2000	Adriana Gersi	CZE	Seda Noorlander	NED	3-6 6-3 6-3
2002	Adriana Serra Zanetti	ITA	Virginia Ruano Pascual	ESP	6-2 2-6 7-5
2002	Adriana Serra Zanetti	ITA	Amy Frazier	USA	6-3 7-6(5)
2002	Adriana Serra Zanetti	ITA	Silvia Farina Elia	ITA	6-2 4-6 6-4
2002	Adriana Serra Zanetti	ITA	Martina Sucha	SVK	6-1 7-5
2003	Adriana Serra Zanetti	ITA	Elena Likhovtseva	RUS	7-6(6) 6-4
2010	Agnes Szavay	HUN	Stephanie Dubois	CAN	6-3 6-2
2007	Agnieszka Radwanska	POL	Varvara Lepchenko	USA	5-7 6-3 6-2
2008	Agnieszka Radwanska	POL	Olga Savchuk	UKR	6-0 6-1
2008	Agnieszka Radwanska	POL	Pauline Parmentier	FRA	7-5 6-4
2008	Agnieszka Radwanska	POL	Svetlana Kuznetsova	RUS	6-3 6-4
2008	Agnieszka Radwanska	POL	Nadia Petrova	RUS	1-6 7-5 6-0
2010	Agnieszka Radwanska	POL	Tatiana Maria	GFR	6-1 6-0

## Code dissection II

---

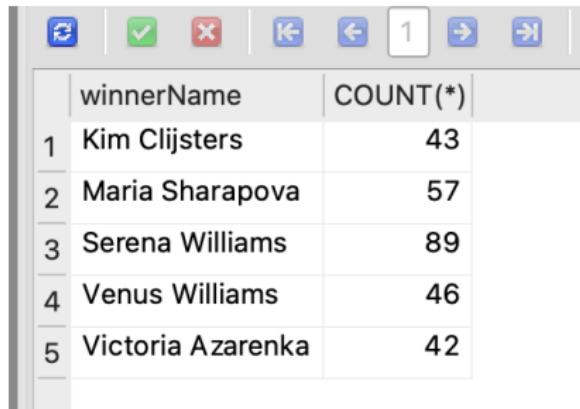
- ▶ The `SELECT` statement says to extract
  - ▶ `winnerName` from each group
  - ▶ The *row count* for each group

	winnerName	COUNT(*)
1	Abigail Spears	2
2	Adriana Gersi	1
3	Adriana Serra Zanetti	5
4	Agnes Szavay	1
5	Agnieszka Radwanska	35
6	Ai Sugiyama	13
7	Aiko Nakamura	5
	...	-

## Code Dissection III

---

Then `HAVING COUNT(*) >= 40` leaves only 5 rows.



A screenshot of a database query results window. At the top, there are several icons: a blue square with a white circle, a green checkmark, a red X, and four arrows pointing left, right, up, and down. To the right of these icons is a page number input field containing the value '1'. Below the header row, there are five data rows, each consisting of a row number, a winner name, and a COUNT(\*) value. The data is as follows:

	winnerName	COUNT(*)
1	Kim Clijsters	43
2	Maria Sharapova	57
3	Serena Williams	89
4	Venus Williams	46
5	Victoria Azarenka	42

# Renaming

---

Our final table is a bit ugly, mostly because `COUNT(*)` is not a very intuitive name for a column.

The screenshot shows a database query interface with the following details:

- Query text:

```
1 SELECT winnerName, COUNT(*) AS numWins
2 FROM WTAResult
3 GROUP BY winnerName
4 HAVING numWins >= 40;
5
6 |
```
- Result table:

	winnerName	numWins
1	Kim Clijsters	43
2	Maria Sharapova	57
3	Serena Williams	89
4	Venus Williams	46
5	Victoria Azarenka	42
- Toolbar buttons: Refresh, Save, Cancel, Run, First, Previous, Next, Last, Grid View, Total rows loaded: 5.

The `AS` statement gives the column a new name, which can then immediately be used in the `HAVING` statement.

## Order at the end

---

The `ORDER BY` statement can be used to put the rows of the output table in suitable order.

The screenshot shows a database query interface with the following details:

- Query Text:**

```
1 SELECT winnerName, COUNT(*) AS numWins
2 FROM WTAResult
3 GROUP BY winnerName
4 HAVING numWins >= 40
5 ORDER BY numWins DESC;
6 |
```
- Grid View:** A table showing the results of the query. The table has two columns: "winnerName" and "numWins". The data is as follows:

	winnerName	numWins
1	Serena Williams	89
2	Maria Sharapova	57
3	Venus Williams	46
4	Kim Clijsters	43
5	Victoria Azarenka	42

- Total rows loaded:** 5

## Group by several factors

---

How many wins did *each player* have *in each year*?

Need to group by both *player* and *year*.

```
SELECT yr, winnerName, COUNT(*) AS numWins  
FROM ATPResult  
GROUP BY yr, winnerName  
HAVING numWins >= 5;
```

- ▶ *Group* rows by (*yr, winnerName*) *combination*
- ▶ *Summarise* each group using *COUNT*
- ▶ *Keep* summary rows with at least 5 wins

# Wins per year

---

```
1 SELECT yr, winnerName, COUNT(*) AS numWins
2 FROM ATPResult
3 GROUP BY yr, winnerName
4 HAVING numWins >= 5;
5
6 |
```

Gr

Total rows loaded: 92

	yr	winnerName	numWins
1	2000	Andre Agassi	7
2	2000	Magnus Norman	5
3	2000	Pete Sampras	5
4	2000	Yevgeny Kafelnikov	6
5	2001	Andre Agassi	7
6	2001	Arnaud Clement	6
7	2001	Patrick Rafter	5
8	2001	Sebastien Grosjean	5

## Another example

---

List each *title* and the *number of cast members* for each movie in the IMDB Top 250.

- ▶ The *title* is only in *titles*
- ▶ Cast members are in *castmembers*

```
SELECT title, COUNT(*) AS numCast
FROM titles JOIN castmembers USING (title_id)
GROUP BY title_id;
```

## What about the bare column?

---

In this query, `title` is a *bare column*.

Didn't we say last week to avoid bare columns?

Firstly, it is not *really* a bare column, because if two rows have the same `title_id` then they must also have the same `title`.

In this situation, there is a *functional dependency* between `title_id` and `title`.

```
SELECT title, COUNT(*) AS numCast
FROM titles JOIN castmembers USING (title_id)
GROUP BY title_id, title;
```

# CITS1402

## Relational Database Management Systems

### Video 17 — Subqueries I

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## When once is not enough

---

The basic `SELECT` command really only allows *one* of everything:

- ▶ Creation of one large input table (`FROM`)
- ▶ One selection from the rows of the input table (`WHERE`)
- ▶ One organisation of the rows into groups (`GROUP BY`)
- ▶ One summary/aggregation of each group (`COUNT`, `SUM` etc)
- ▶ One final pass through the summary rows (`HAVING`)

In general, when one row is being processed, it is hard to use information about the *other rows* in the same table.

## A two-step process

---

If we want to use the result of one query as part of another query, it seems to need two steps.

- ▶ *Find the length* of the longest match

```
SELECT MAX(minutes)  
FROM ATPResult;
```

(The result is 353)

- ▶ *Use the value* in a second query

```
SELECT yr, winnerName, loserName, score  
FROM ATPResult  
WHERE minutes = 353;
```

## Nested Queries

---

A *nested query* is a query that involves *another query* as one of its component parts.

```
SELECT yr, winnerName, loserName, score  
FROM ATPResult  
WHERE minutes = (SELECT MAX(minutes) FROM MatchResult);
```

This query involves a *subquery* to find the details of the longest men's singles match in the Australian Open database.

## Inner query

---

```
SELECT tournYear, winnerName, loserName, score  
FROM ATPResult  
WHERE minutes = (SELECT MAX(minutes) FROM ATPResult);
```

If we ran the *inner query* on its own, we would get a table with 1 row and 1 column, containing the value 353.

This is treated as a *scalar value* (i.e., just as a number).

Any subquery that returns a *single value* is called a *scalar* subquery.

## Outer query

---

The *outer query* then becomes

```
SELECT yr, winnerName, loserName, score  
FROM ATPResult  
WHERE minutes = (      353      );
```

## Types of subquery

---

How a subquery can be manipulated depends on the type of results that it produces:

- ▶ A *scalar* subquery produces a *single value* (that is, a table with one row and one column) as a result
- ▶ A *column* subquery produces a single column as a result
- ▶ A *row* subquery produces a single row as a result
- ▶ A *table* subquery produces an entire table as a result

## Scalar subqueries

---

The result of a scalar subquery can be used essentially anywhere that a single value can be used, e.g. you can make comparisons with <, >, =, <> and so on.

Sometimes a scalar subquery is just used to find an unknown value from another table:

```
SELECT *
FROM City
WHERE CountryCode = (SELECT Code
                      FROM Country
                      WHERE name = 'Australia');
```

# Equivalent to a join

A subquery like this equivalent to a **JOIN**.

```
SELECT City.* FROM City, Country  
WHERE CountryCode = Code  
AND Country.name = 'Australia';
```

The screenshot shows a database query interface with the following details:

- Query Tab:** The tab is selected, showing the SQL code:

```
1 SELECT City.* FROM City, Country  
2 WHERE CountryCode = Code  
3 AND Country.name = 'Australia';
```
- Grid view:** The results are displayed in a grid format. The columns are labeled ID, Name, CountryCode, and pop.
- Data:** The grid contains 5 rows of data:

ID	Name	CountryCode	pop
1	130	Sydney	AUS
2	131	Melbourne	AUS
3	132	Brisbane	AUS
4	133	Perth	AUS
5	134	Adelaide	AUS

Total rows loaded: 14

What does **SELECT City.\*** do?

## IN OR NOT IN

---

If a subquery returns *one column*, then it is viewed as a *set of values* that can be used with the special operators `IN` or `NOT IN`.

Which players *competed in* the AO but *never won* any matches?

To answer this, we need to find a player who *never appears* in the `winnerName` column.

## Consistent losers

---

```
SELECT DISTINCT loserName  
FROM ATPResult  
WHERE loserName  
    NOT IN (SELECT DISTINCT winnerName  
            FROM ATPResult);
```

	loserName
1	Mariano Puerta
2	Jeff Tarango
3	Noam Okun
4	Jim Courier
5	Dejan Petrovic
6	Hernan Gumy

## Analysis of this query

---

The *inner query* just produces a bunch of names of all the players who have ever won any single match.

```
SELECT DISTINCT winnerName  
FROM ATPResult;
```

9	Richard Krajicek
10	Nicolas Escude
11	Leander Paes
12	Andreas Vinciguerra
13	Fredrik Jonsson
14	Hicham Arazi
15	Fernando Vicente
16	Todd Martin
17	Pete Sampras
18	Mikael Tillstrom
19	Marc Rosset
20	Wayne Black

The `SELECT DISTINCT` removes any duplicate names from the list.

## The list of all the losers

---

The players who have *never won* a match are those whose name

- ▶ *does* appear in the column `loserName`, and
- ▶ *does not* appear in the column `winnerName`

```
SELECT DISTINCT loserName
FROM ATPResult
WHERE loserName
    NOT IN (SELECT DISTINCT winnerName
              FROM ATPResult) ;
```

In order to ensure that each player's name is listed only once, we use `SELECT DISTINCT` to remove duplicates.

## Most persistent

---

Some persistent players come back for the Australian Open year-after-year despite *always losing* in the first round.

Who is the *most persistent* player in the database?

To find this out we need to find the *number of matches* that have been played by each of the players who has *never won*.

We need to combine aggregation with nested queries.

# Aggregation

---

Instead of taking the list of losing players and just removing duplicates, we can instead *count the number of occurrences* of each name.

```
SELECT loserName, COUNT(*) AS numLosses
FROM ATPResult
WHERE loserName
    NOT IN (SELECT DISTINCT winnerName
              FROM ATPResult)
GROUP BY loserName
ORDER BY numLosses DESC;
```

# Who are they?

---

	loserName	numLosses
1	Potito Starace	7
2	Marcos Daniel	5
3	Marco Cecchinato	5
4	Laslo Djere	5
5	Kenneth Carlsen	5
6	Horacio Zeballos	5
7	Cedrik Marcel Stebe	5

Note: Starace was banned for life by the Italian Tennis Federation for gambling offences.

## Sample Questions

---

- ▶ Write a single SQL query that will list the match details for all ATP matches that were more than 2 hours longer than the overall average ATP match length.
- ▶ Write a single SQL query that will list all female players, once each, who are taller than the average male player.

CITS1402  
Relational Database Management Systems

Video 18 — Entity Relationship Diagrams II

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# What is an ERD?

---

Ask ten database designers and you'll get ten different answers.

An ERD is a visual way of representing

- ▶ The *entities* being represented in the database
- ▶ The *attributes* of those entities
- ▶ The *relationships* between those entities
- ▶ Certain *constraints* on all of the above

## ERDPlus conventions

---

- ▶ *Entities* are represented by *rectangles*
- ▶ *Attributes* are represented by dangling *ellipses*
- ▶ *Relationships* are represented by *diamonds*

(But remember that ultimately, an RDBMS only has *tables*.)

# A university database

---

The university needs to maintain a student grade database with the following properties:

- ▶ Students have a name and a unique student number
- ▶ Units have a name and a unique course code
- ▶ Students enrol in units in a particular year and get a mark

## Entity Sets

---

Entity sets are the collections of objects about which the database must keep information.

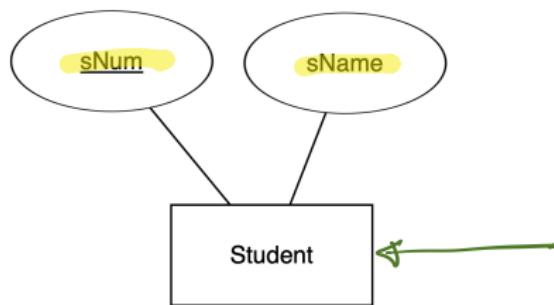
For example, the collection of *students* and the collection of *units* are each entity sets.

Each entity set has a collection of *attributes* which define what information is maintained about each individual entity.

# Student

---

A **Student** has a *student number*, and a *name*.



The *attributes* are shown in ellipses hanging off the rectangle.

## Entities are tables

---

snum	sname
1	"Amy"
2	"Bill"

When this diagram is turned into an actual relational schema.

- ▶ Each *entity* corresponds to *table*
- ▶ Its *attributes* are the *columns* of the table

## Create the table

---

The SQLite code to create the table is

```
CREATE TABLE Student (
    sNum INTEGER,
    sName TEXT);
```

*sName VARCHAR(64)*

- ▶ The **name of the entity (set)** is the **table name**
- ▶ The attributes of the entity set are the columns
- ▶ Each column has now been given a **type**

## But there's more

---

a field/column  
that does not  
allow duplicates.

In the ERD the **sNum** attribute was underlined.

This indicates that this attribute is a **key** and therefore **uniquely identifies** a row in this table.

The “**business logic**” of a university tells us that different students cannot have the same **sNum**.

We should **tell SQLite** about this.

## Primary Key

---

When the table is created, we can tell SQLite that the `sNum` field is to be a key.

```
CREATE TABLE Student (
    sNum TEXT,
    sName TEXT,
    PRIMARY KEY (sNum)
);
```

SQLite will then *prevent* any operations that would violate this.

# Data Integrity

---

Declaring columns to be keys is one of the first layers of defence of data integrity.

The screenshot shows a SQL query window with the following content:

```
7 INSERT INTO Student VALUES(1, "Amy");
8 INSERT INTO Student VALUES(1, "Bill");
```

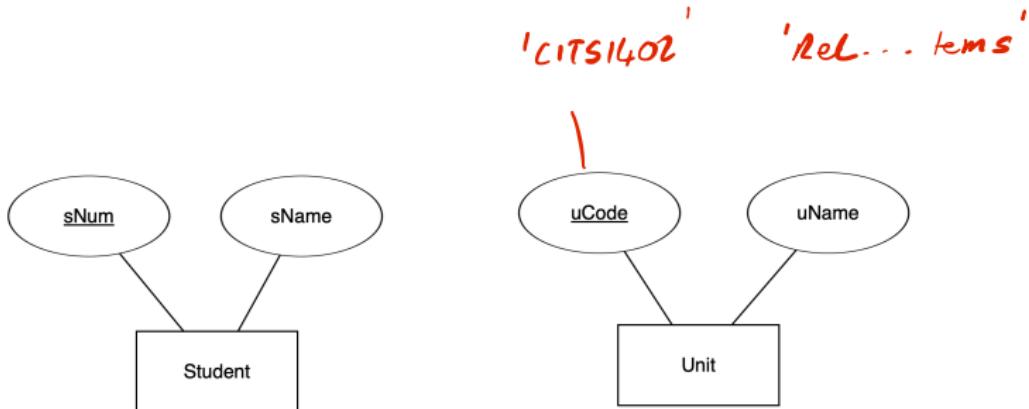
The second query (line 8) has the value '1' highlighted and circled in red. The status bar at the bottom of the window displays the following log entries:

- [18:31:40] Query finished in 0.003 second(s).
- [18:32:12] Query finished in 0.002 second(s). Rows affected: 1
- [18:32:18] Error while executing SQL query on database 'University': UNIQUE constraint failed: Student.sNum

This prevents simple data-entry errors that could have far-reaching consequences.

# Units are similar

---



## Relationships

---

Students *enrol* in units, and so each individual enrolment involves:

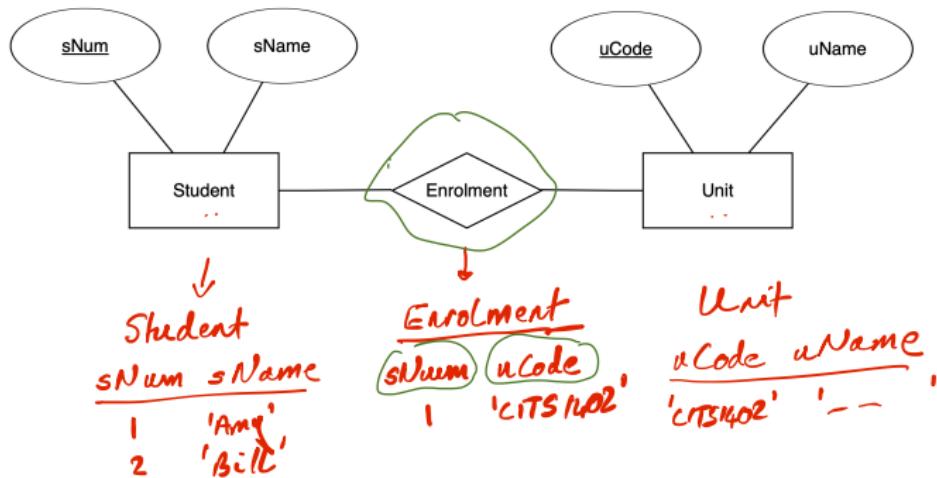
- ▶ One student
- ▶ One unit

Enrolling is therefore a *relationship* between a particular student and a particular unit.

# Relationships

---

A relationship is represented by a *diamond*.



# Implementing a relationship

---

A *relationship* in an ERD will *also* become a table in the database.

```
CREATE TABLE Enrolment (
    sNum INTEGER,
    uCode TEXT,
);
```

In this case, each row of the table contains a student number and a unit code.

## Relationship Attributes

---

In reality, we need to store *more information* about enrolments



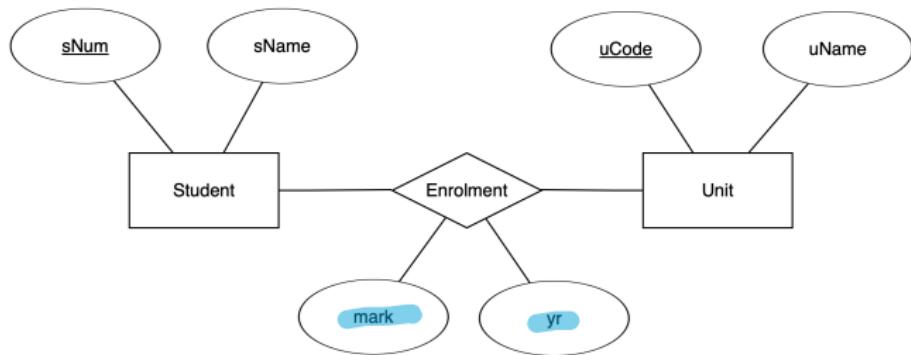
Amy took Databases in 2018 and got a mark of 45%

Amy took Databases in 2019 and got a mark of 69%

A *mark* is not an attribute of a *Student*, nor is it an attribute of a *Unit* — it is really an attribute of the *combination* of a *mark* and a student.

# Relationship attributes

---



## In the schema

---

```
CREATE TABLE Enrolment (
    sNum INTEGER,
    uCode TEXT,
    yr INT,
    mark INT
);
```

	'CITS1402'	2019	45
	'CITS1402'	2019	69
:			

## Composite keys

---

A *composite key* is a key that involves some *combination* of attributes.

Does it make sense for there to be

- ▶ Multiple rows with the same sNum? Yes
- ▶ Multiple rows with the same sNum *and* the same uCode? Yes
- ▶ Multiple rows with the same sNum, uCode *and* yr?

| 'CITS1402' 2018 ↗  
| 'CITS1402' 2018

## In the schema

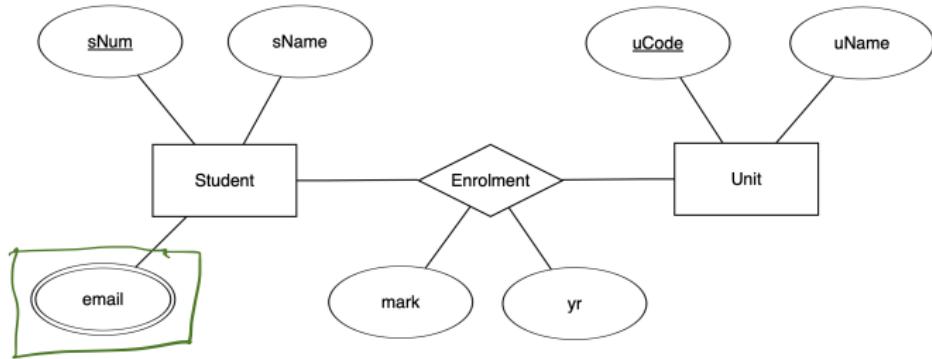
---

```
CREATE TABLE Enrolment (
    sNum INTEGER,
    uCode TEXT,
    yr INT,
    mark INT,
    PRIMARY KEY (sNum, uCode, yr)
);
```

# Multi-valued attributes

---

A *multi-valued attribute* can take *multiple values*. .



A student can have several email addresses:

# Implementing multi-valued attributes

---

Multi-value attributes cannot be directly implemented in a relational database, but we can simulate this as follows:

- ▶ Use some predetermined **TEXT** format (e.g. comma-separated)  
- s1@uwa.edu.au, x123@gmail.com,
- ▶ Use a **separate table** for email addresses

```
CREATE TABLE emailList (
    sNum INTEGER,
    email TEXT
);
```

## Using the list

---

```
INSERT INTO emailList VALUES(1, "amy@gmail.com");
INSERT INTO emailList VALUES(1, "pres@guild.uwa");
INSERT INTO emailList VALUES(1, "s1@uwa.edu.au");
```

We want to email all the students currently taking CITS1402.

```
SELECT email
FROM Enrolment
JOIN
emailList ON Enrolment.sNum = emailList.sNum
WHERE Enrolment.uCode = "CITS1402" AND
Enrolment.yr = 2020;
```

# CITS1402

## Relational Database Management Systems

### Video 19 — Subqueries II

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



THE UNIVERSITY OF  
**WESTERN**  
AUSTRALIA

# Length of long movies

How *long* is the longest IMDB Top-250 movie(s) each year?

```
SELECT premiered, MAX(runtime_minutes)
FROM titles
GROUP BY premiered;
```

The screenshot shows a database interface with a SQL query editor at the top and a results grid below. The query is:

```
1 SELECT premiered, MAX(runtime_minutes)
2 FROM titles
3 GROUP BY premiered;
```

The results grid has two columns: 'premiered' and 'MAX(runtime\_minutes)'. The data is as follows:

	premiered	MAX(runtime_minutes)
1	1921	68
2	1925	95
3	1927	153
4	1931	117
5	1934	105
6	1936	87
7	1939	238
8	1940	130
9	1941	119
10	1942	102

## Names of long movies

---

What are the names of these long movies?

```
SELECT premiered, title, runtime_minutes
FROM titles
WHERE (title, runtime_minutes) IN
(SELECT premiered, MAX(runtime_minutes)
FROM titles
GROUP BY premiered);
```

The table in the subquery is called a *derived table*.

# The output

---

```
1 SELECT premiered, title, runtime_minutes
2 FROM titles
3 WHERE (premiered, runtime_minutes) IN
4   (SELECT premiered, MAX(runtime_minutes)
5    FROM titles
6    GROUP BY premiered)
7 ORDER BY premiered;
8 |
```

Grid view Form view

Total rows loaded: 84

	premiered	title	runtime_minutes
1	1921	The Kid	68
2	1925	The Gold Rush	95
3	1927	Metropolis	153
4	1931	M	117
5	1934	It Happened One Night	105
6	1936	Modern Times	87
7	1939	Gone with the Wind	238

## How does it work?

---

First, the subquery is run and its output (which is a 2-column table) saved

Next, the main query the rows of **titles**, one at a time and checks the **WHERE** condition:

If the pair (`premiered`, `runtime_minutes`) appears in the saved table from the subquery, then this row refers to one of the longest movies from that year and so the row is kept.

Finally, the **SELECT** statement extracts the year, title and length of the movie to produce as output.

## Correlated or not

---

Subqueries can either be *uncorrelated* or *correlated*.

- ▶ Uncorrelated

A subquery is *uncorrelated* if it is a complete query that can be run in isolation from the outer query.

Uncorrelated subqueries do not refer to any columns from the tables in the outer query.

- ▶ Correlated

A subquery is *correlated* if it *does* involve columns from the tables in the outer query, and therefore does not form a complete query in its own right.

## Uncorrelated subquery

---

This subquery is uncorrelated:

```
SELECT title, runtime_minutes
FROM titles
WHERE runtime_minutes >
(SELECT AVG(runtime_minutes)
 FROM titles);
```

It is logically equivalent to running the inner query *once*, replacing the inner query with the result, and then running the outer query.

## Correlated subquery

---

In a *correlated subquery* the inner query uses a *value* from the outer query.

```
SELECT T1.premiered, T1.title, T1.runtime_minutes
FROM titles T1
WHERE T1.runtime_minutes >
  (SELECT AVG(T2.runtime_minutes)
   FROM titles T2
   WHERE T2.premiered = T1.premiered);
```

Here there are two instances of table `titles`, called T1 and T2 and the inner query uses a value from T2.

## Logically speaking

---

Visualise this as follows:

- ▶ For each row of `titles T1`  
('tt0111161', 'The Shawshank Redemption', 1994, 142)
- ▶ The value of `T1.premiered` is now inserted into the *inner query*:

```
(SELECT AVG(T2.runtime_minutes)
 FROM titles T2
 WHERE T2.premiered = 1994);
```

- ▶ The subquery is run, returning a value 122.5
- ▶ The `WHERE` clause of the outer query is now evaluated  
As  $142 > 122.5$ , the `WHERE` condition is true, and so this first row of `T1` is part of the output.

## Many runs

---

In principle, the inner query is run once for *every single row* of the outer table.

The inner queries are *slightly different* each time — for every row of the outer table, the inner query is only “completed” by inserting particular values determined by the row currently being considered.

If the system is unable to find a way to optimize this query, it could potentially be very time-consuming.

## Exists

---

A very common sub-query is to just establish the *existence* or *non-existence* of a particular row.

For example, a student has the highest mark in a class if there *does not exist* a student with a higher mark in the same class

This is accomplished with **EXISTS** and **NOT EXISTS**.

## Example schema

---

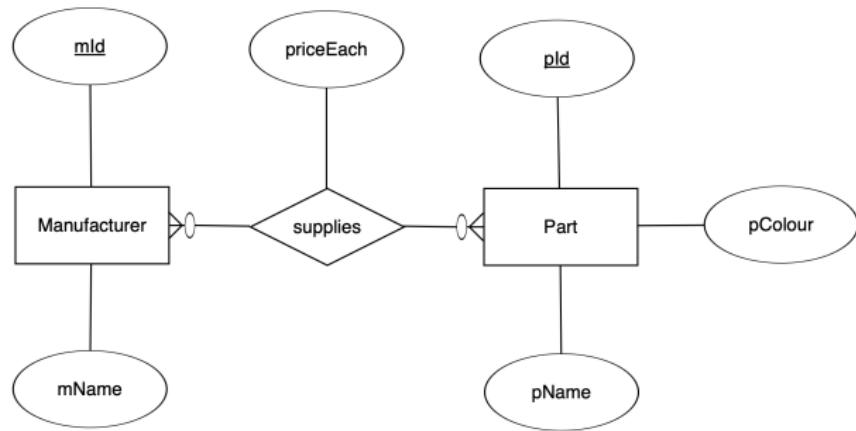
We will use a schema regarding suppliers of products.

```
CREATE TABLE Manufacturer (
    mId INTEGER PRIMARY KEY,
    mName TEXT );
```

```
CREATE TABLE Part (
    pId INTEGER PRIMARY KEY,
    pName TEXT,
    pColour TEXT);
```

# The ERD

---



## The catalogue

---

Each year a catalogue is produced listing the cost that each manufacturer is charging for various parts.

```
CREATE TABLE supplies (
    mId  INTEGER,
    pId  INTEGER,
    priceEach REAL);
```

So `supplies` is a *relationship* between manufacturers and parts, and `priceEach` is a *relationship attribute*.

## EXISTS and NOT EXISTS

---

Which parts are no longer being manufactured?

```
SELECT P.pName FROM Part P
WHERE NOT EXISTS (SELECT *
                   FROM supplies S
                   WHERE S.pId = P.pId);
```

(Company must now find another manufacturer or start building their own parts.)

## Which parts are supplied only by Acme?

---

Find the *names* of the parts supplied *only* by Acme.

```
SELECT P.pName
FROM Manufacturers M JOIN supplies S USING (mId)
    JOIN Part P USING (pId)
WHERE M.mName = 'Acme'
AND NOT EXISTS (SELECT *
    FROM supplies S2
    WHERE S2.mId <> S.mId
    AND S2.pId = P.pid);
```

## More than one way to skin a cat

---

Pretty much anything that can be done with `EXISTS` and `NOT EXISTS` can be done with `IN` and `NOT IN`.

There can be efficiency differences, but given that SQLite (any implementation of SQL) optimises queries, it is not necessarily obvious which is faster in any given situation.

# CITS1402

## Relational Database Management Systems

### Video 20 — Relational Algebra II

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Relational Algebra

---

A legal expression in *relational algebra* is built from

- ▶ Variables representing *relations*  $R, S, T$
- ▶ Set-theoretic operators  $\cup, \cap$  *boutie*.
- ▶ “Relational” operators  $\sigma, \pi, \bowtie$

composed according to the rules of relational algebra.

Every *legal expression* in relational algebra determines a *relation*.

# Operators

---

union compatible  
= same number  
of columns



- ▶ Set-theoretic operators

The *union* of two relations  $R \cup S$ , the *intersection* of two relations  $R \cap S$  and the *difference* of two relations  $R - S$ .

- ▶ Relational operators

The *projection* operator  $\pi$ , the *selection* operator  $\sigma$  and the *join* operator  $\bowtie$ .

## Set Theory Operators

---

Relational algebra permits the use of the standard set operations:

- ▶ Union ( $\cup$ )  
If  $R$  and  $S$  are *union-compatible*, then  $R \cup S$  is the set of tuples in *either*  $R$  or  $S$ .
- ▶ Intersection ( $\cap$ )  
If  $R$  and  $S$  are union-compatible, then  $R \cap S$  is the set of tuples in *both*  $R$  and  $S$ .
- ▶ Set Difference ( $-$ )  
If  $R$  and  $S$  are union-compatible then  $R - S$  is the set of tuples in  $R$  that are *not in*  $S$

## Example

relation is R  
two columns A, B

Suppose that  $R(A, B)$  and  $S(C, D)$  are relations as follows:

A	B
1	2
4	2
3	3

Relation R

C	D
0	1
1	3
2	4
3	3

Relation S

We can write

$$R = \{(1, 2), (4, 2), (3, 3)\}$$

## In SQLite

---

In SQLite,

- ▶ Use **UNION** for union
- ▶ Use **INTERSECT** for intersection
- ▶ Use **EXCEPT** for set difference

# Examples

---

```
1 SELECT * FROM R  
2 UNION  
3 SELECT * FROM S;
```

The screenshot shows two tables, R and S, displayed side-by-side. Table R has columns A and B, with data: (1, 0), (2, 1), (3, 1), (4, 2), (5, 3), (6, 4). Table S has columns A and B, with data: (1, 1), (2, 4). A red brace on the left groups rows 1 through 5 under the heading '6.', indicating they are part of the union result.

	A	B
1	0	1
2	1	2
3	1	3
4	2	4
5	3	3
6	4	2

```
1 SELECT * FROM R  
2 INTERSECT  
3 SELECT * FROM S;
```

The screenshot shows the result of an INTERSECT query. It contains two rows: (1, 3) from table R and (1, 3) from table S. A red brace on the left groups these two rows together, indicating they are the intersection of the two datasets.

	A	B
1	3	3

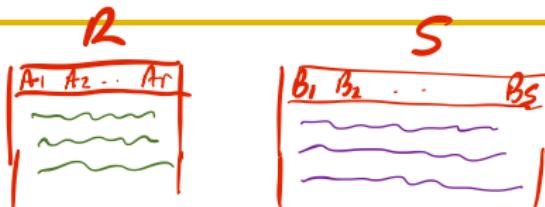
```
1 SELECT * FROM R  
2 EXCEPT  
3 SELECT * FROM S;
```

The screenshot shows the result of an EXCEPT query. It contains two rows from table R that are not present in table S: (1, 0) and (2, 1). A red brace on the left groups these two rows together, indicating they are the result of the set difference operation.

	A	B
1	1	2
2	4	2

# Cartesian Products

---



If  $R$  has  $r$  columns and  $S$  has  $s$  columns, then their *Cartesian Product*

$$R \times S$$

has  $r + s$  columns.



Each row of  $R \times S$  has the property that

- ▶ The first  $r$  columns are a row of  $R$
- ▶ The last  $s$  columns are a row of  $S$

# In SQLite

---

```
1 SELECT * FROM R , S;
```

	A	B	C	D	Total
1	1	2	0	1	
2	1	2	1	3	
3	1	2	2	4	
4	1	2	3	3	
5	4	2	0	1	
6	4	2	1	3	
7	4	2	2	4	
8	4	2	3	3	
9	3	3	0	1	
10	3	3	1	3	
11	3	3	2	4	
12	3	3	3	3	

```
1 SELECT * FROM R JOIN S;
```

	A	B	C	D	Total
1	1	2	0	1	
2	1	2	1	3	
3	1	2	2	4	
4	1	2	3	3	
5	4	2	0	1	
6	4	2	1	3	
7	4	2	2	4	
8	4	2	3	3	
9	3	3	0	1	
10	3	3	1	3	
11	3	3	2	4	
12	3	3	3	3	

## Joins

---

Suppose that  $c$  is a *boolean function* that may involve the attributes of both  $R$  and  $S$ .

Then

$$R \bowtie_c S$$

is defined to be

$$\underline{\sigma}_c(R \times S).$$

In other words a *join* is the result of selecting certain rows from the Cartesian product.

# Examples

$A = 1 \quad C = 3$   
AND

$R \bowtie S$   
 $A=1$

	A	B	C	D	
1	1	2	0	1	
2	1	2	1	3	
3	1	2	2	4	
4	1	2	3	3	
5	4	2	0	1	
6	4	2	1	3	
7	4	2	2	4	
8	4	2	3	3	
9	3	3	0	1	
10	3	3	1	3	
11	3	3	2	4	
12	3	3	3	3	

	A	B	C	D	
1	1	2	0	1	
2	1	2	1	3	
3	1	2	2	4	
4	1	2	3	3	
5	4	2	0	1	
6	4	2	1	3	
7	4	2	2	4	
8	4	2	3	3	
9	3	3	0	1	
10	3	3	1	3	
11	3	3	2	4	
12	3	3	3	3	

	A	B	C	D	
1	1	2	0	1	
2	1	2	1	3	
3	1	2	2	4	
4	1	2	3	3	
5	4	2	0	1	
6	4	2	1	3	
7	4	2	2	4	
8	4	2	3	3	
9	3	3	0	1	
10	3	3	1	3	
11	3	3	2	4	
12	3	3	3	3	

$R \bowtie A=1 S$

$R \bowtie (A=1 \wedge C=3) S$

$R \bowtie A < D S$

SELECT \* FROM R, S  
WHERE A < D;

## Natural Join

---

The *natural join* of two relations  $R$  and  $S$ , denoted

$$R \bowtie S$$

is the join where the condition is “every column of  $R$  and  $S$  with the *same name* must match”.

In this *special case* we leave out the  $c$  in  $R \bowtie_c S$ .

# In SQLite

```
CREATE TABLE T (A INT, D INT);
INSERT INTO T SELECT C,D FROM S;

SELECT * FROM R NATURAL JOIN T;
```

The screenshot shows the SQLite Studio interface with two tables, R and T, displayed in a grid. Table R has columns A and B, with data: (1, 2), (2, 3). Table T has columns A and D, with data: (1, 3), (3, 1). A natural join is performed on column A, resulting in the following joined data:

	R.A	R.B	T.A	T.D
1	1	2	1	3
2	2	3	3	1

↑ salitestudio

$R(A, B)$   
 $S(C, D)$   
 $T(A, D)$

# Natural Join

---

We could have used `NATURAL JOIN` in the Classic Models database.

Both `orderdetails` and `products` use `productCode` to identify individual products.

The screenshot shows a database interface with a SQL query editor and a results grid. The query is:

```
6 SELECT * FROM
7 orderdetails NATURAL JOIN products
```

The results grid displays data from both tables joined on the `productCode` column. Red arrows point from the highlighted `productCode` columns in the SQL code to the corresponding columns in the results grid.

	orderNum	productCode	quantity	priceEach	orderLine	productCode	productName	prod
1	10100	S18_1749	30	136	3	S18_1749	1917 Grand Touring Sedan	Vinta
2	10100	S18_2248	50	55.09	2	S18_2248	1911 Ford Town Car	Vinta
3	10100	S18_4409	22	75.46	4	S18_4409	1932 Alfa Romeo 8C2300 Spider Sport	Vinta
4	10100	S24_3969	49	35.29	1	S24_3969	1936 Mercedes Benz 500k Roadster	Vinta
5	10101	S18_2325	25	108.06	4	S18_2325	1932 Model A Ford J-Coupe	Vinta

## Problems with NATURAL JOIN

---

The natural join matches *all same-name columns*.

At the moment `orderdetails` and `products` have only one same-name column, namely `productCode`.

But what if someone comes along and later adds another column to `products` with the name `orderNumber`?



## An alternative

---

Rather than leaving it to the system to match up the correct column names, it can be done explicitly using **USING**.

```
SELECT *  
FROM orderDetails JOIN products  
USING (productCode);
```

on orderDetails.productCode

= products.productCode;

This is more robust and more readable.

# Past Exam Question

---

(Remember that in relational algebra, a relation is a **set** of tuples, so cannot have duplicates.)

1. Consider a relation  $R(A, B, C)$  containing the following tuples

A	B	C
1	2	4
1	2	3
3	3	1

How many tuples are in the relation

$$\pi_{A,B}(R) \times \pi_{A,C}(R)$$

- (a) 3
- (b) 5
- (c) 6
- (d) 9

# Past Exam Question

---

---

2. How many tuples are in the relation

$$\pi_{A,B}(R) \bowtie \pi_{B,C}(R)$$

where  $R$  is the same relation as in Question 1.

- (a) 3
  - (b) 5
  - (c) 6
  - (d) 9
-