

# 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.

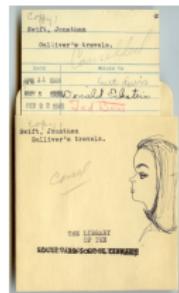
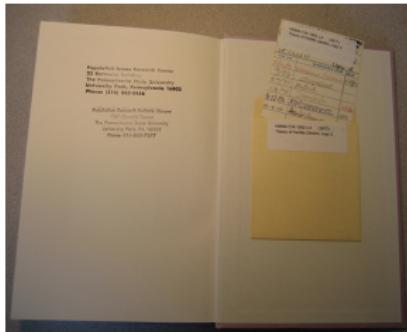


Cards are *alphabetically sorted* to allow readers to quickly find available books.

## 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**.

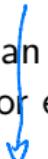
In other words, the attributes define **what type of data** is to be stored for each entity.

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	182	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.

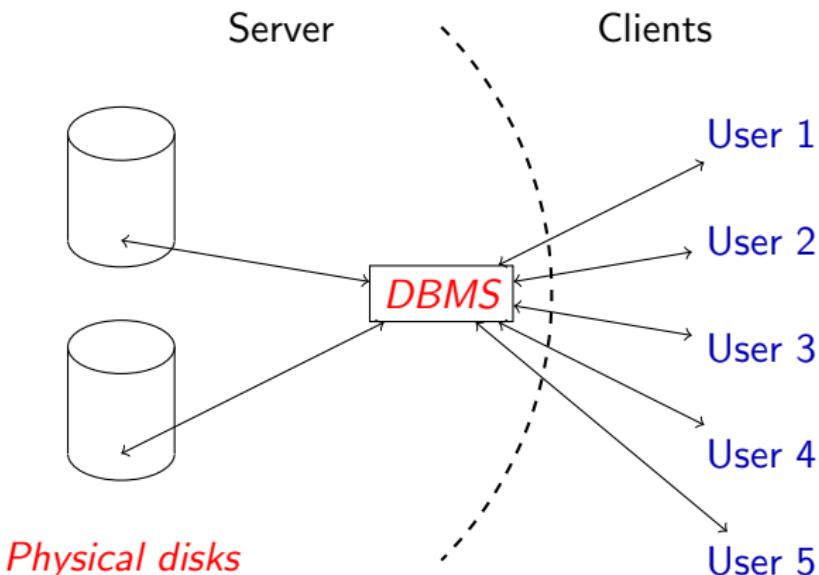
The most commonly used (Oracle, MySQL, PostgreSQL and Microsoft SQL Server) all use the *client/server* model.

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

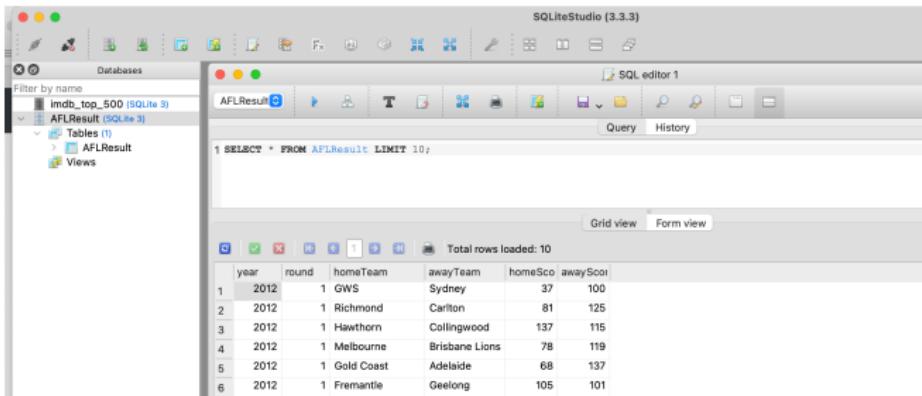
- ▶ *Starts* sqlite3 session in a *terminal window*
- ▶ *Attaches* a database file using a *dot command*
- ▶ Repeatedly *types* either SQL or dot commands
- ▶ *Quits* the session with .quit

```
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
```

## 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 SQLite database interface titled "sqliteWork". A query has been run: "SELECT \* FROM City, Country;". The results are displayed in a grid view. The first column is labeled "id" and contains values from 1 to 12. The second column is labeled "name" and contains the value "Kabul" repeated 12 times. The third column is "countryCode" and contains "AFG" repeated 12 times. The fourth column is "population" and contains "1780000" repeated 12 times. The fifth column is "code" and contains "ABW" for rows 1-2, "AFG" for rows 3-11, and "ATA" for row 12. The sixth column is "name1" which lists various countries: Aruba, Afghanistan, Angola, Anguilla, Albania, Andorra, Netherlands Antilles, United Arab Emirates, Argentina, Armenia, American Samoa, and Antarctica. The seventh column is "capital" and the eighth column is "continent". Rows 1 through 11 have identical capital and continent values (129, North America), while row 12 has "NULL" for the capital and "Antarctica" for the continent. A red box highlights the first row, and a blue box highlights the entire row 12.

Total rows loaded: 974881							
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

---

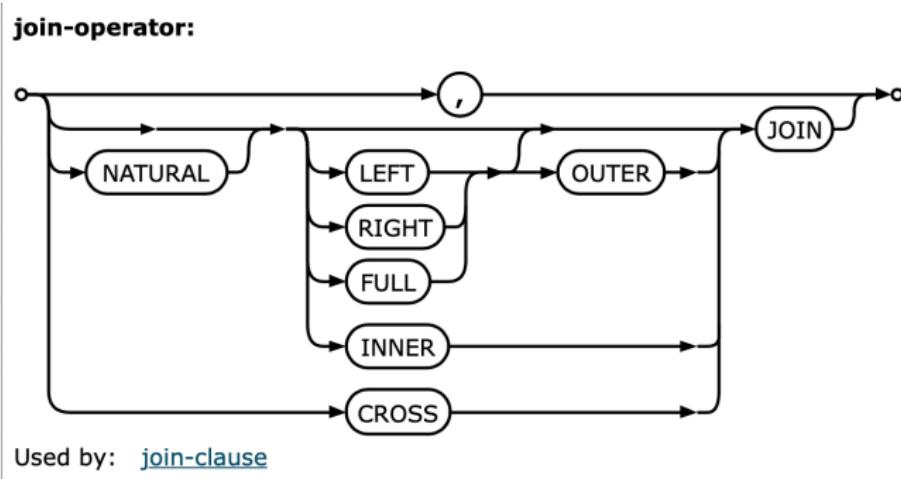
We have to distinguish between the **name** that came from **City** and the ***other name*** that originally came from **Country**.

```
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



THE UNIVERSITY OF  
**WESTERN**  
**AUSTRALIA**

# 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

---

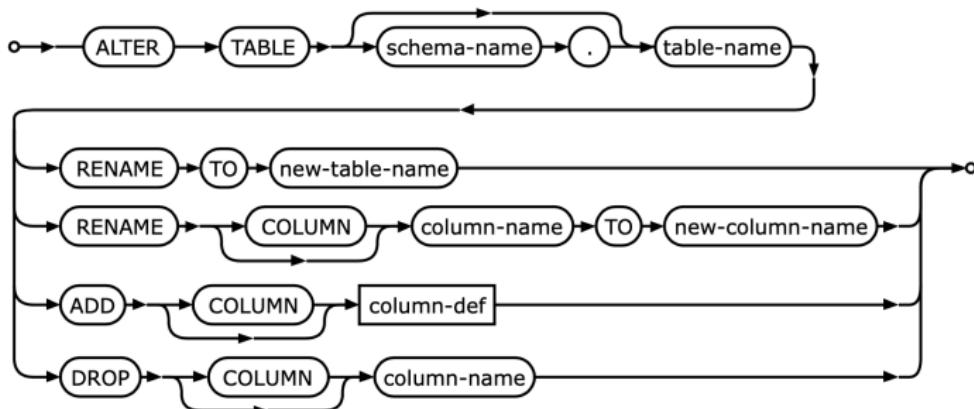
The major DDL statements all contain the word **TABLE**.

```
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



THE UNIVERSITY OF  
**WESTERN**  
**AUSTRALIA**

## 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



THE UNIVERSITY OF  
**WESTERN**  
AUSTRALIA

# 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.

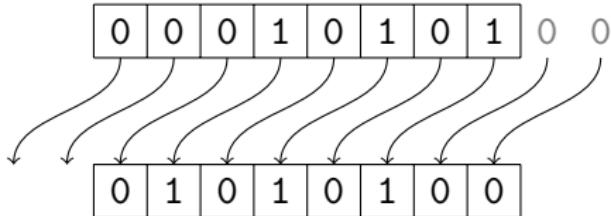
# Bitwise operators

---

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 contains 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 table 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, which corresponds to the result of the query, is highlighted in gray. The row contains the values: 2011, Francesca Schiavone, Svetlana Kuznetsova, and 6-4 1-6 16-14. Above the grid, a message says 'Total rows loaded: 1'. The bottom of the grid has navigation icons for sorting and filtering.

# 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 (refresh, save, etc.) and a message: "Total rows loaded: 1". A single row of data is shown:

AVG(minutes)
1 148.22889498970488

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



THE UNIVERSITY OF  
**WESTERN**  
**AUSTRALIA**

# 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

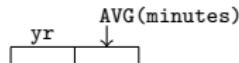
Total rows loaded: 23

# 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	Total n
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 toolbar includes standard icons for opening files, saving, and executing queries. Below the toolbar, the tabs "Query" and "History" are visible, with "Query" being the active tab.

The query itself is:

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

Below the query, there are two tabs: "Grid view" (which is selected) and "Form view". A message "Total rows loaded: 5" is displayed above the grid. The grid displays the following data:

	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

---

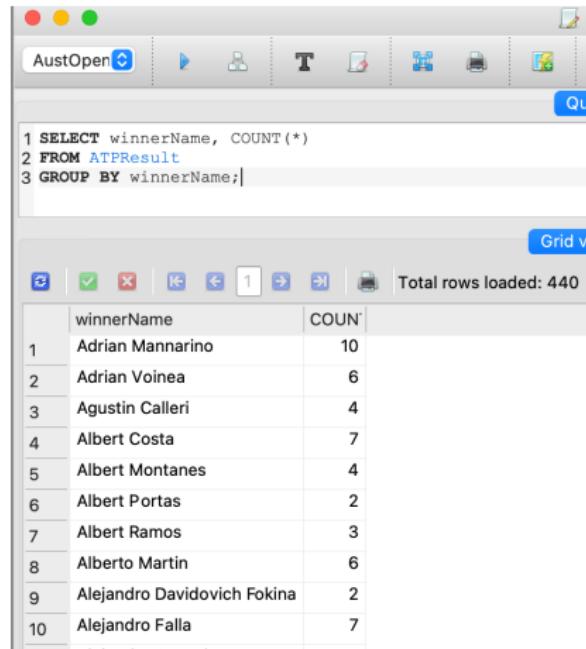
How many total wins has each male player had?

- ▶ 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

---

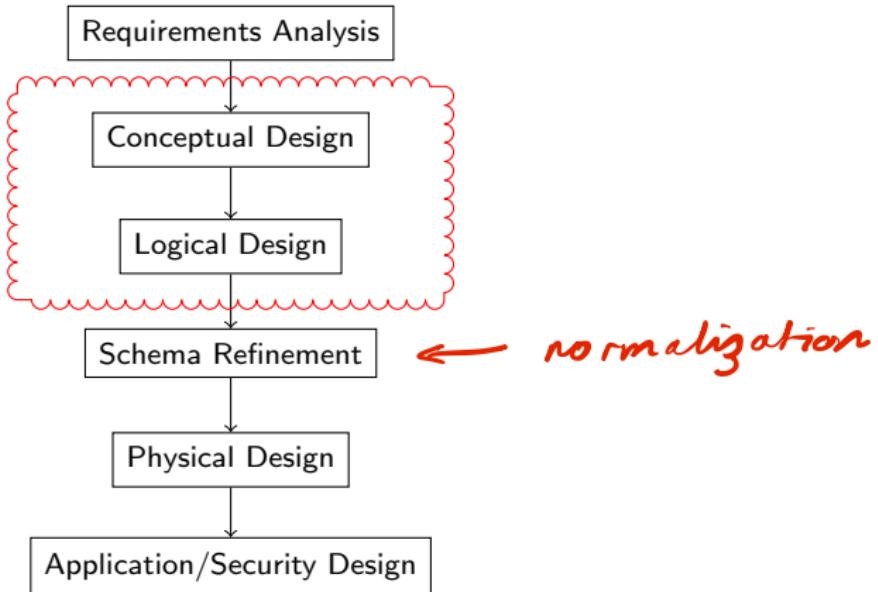
Ramakrishnan & Gehrke identify six steps in *designing* a database

- ▶ 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

---

- ▶ 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.
- Student  
Course  
Unit -*
- “what  
things  
are like”*
- ✓ how to  
do this*

## 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:

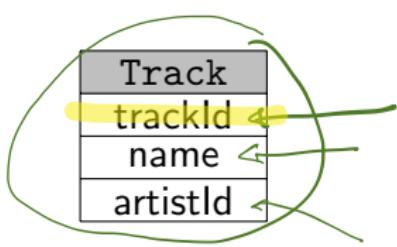
- ▶ New or inadequately-analysed use cases
- ▶ Additional or enhanced functionality

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

# Diagramming

---

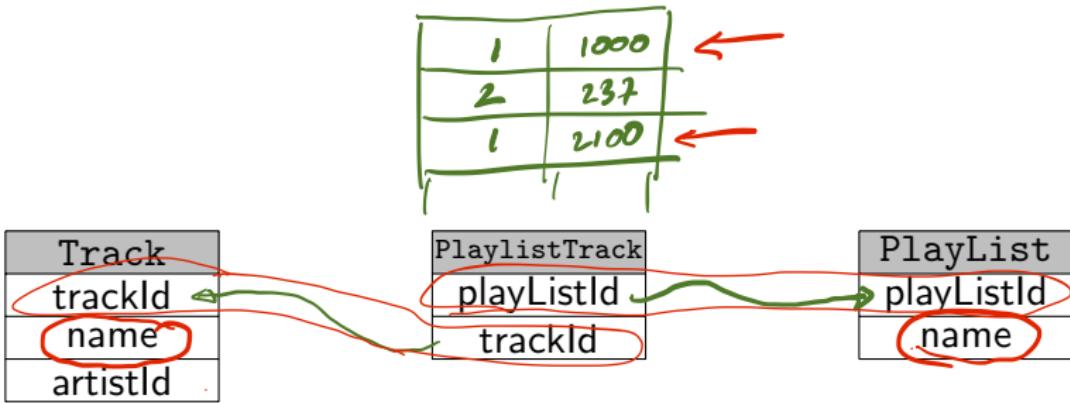
When *designing* a new database or *understanding* an existing database it is often convenient to *draw pictures*.



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

# Diagramming

---



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

## The bridge table

---

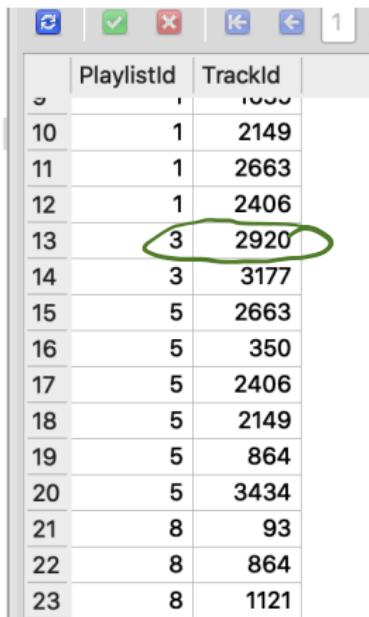
A **Track** can be on many **PlayList**s and a **PlayList** can contain many **Tracks**.

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.

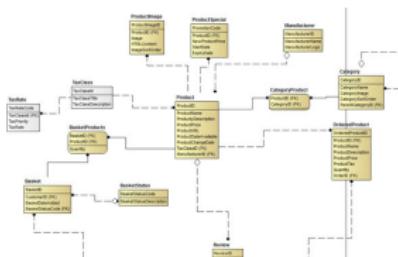
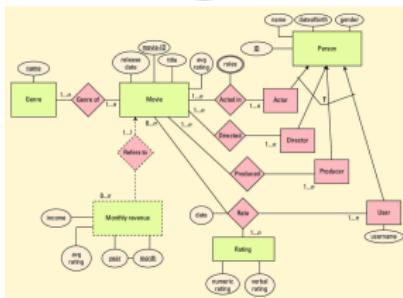
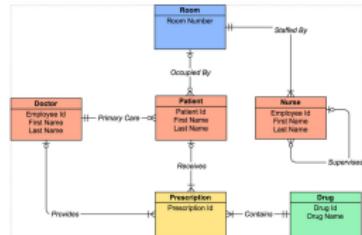
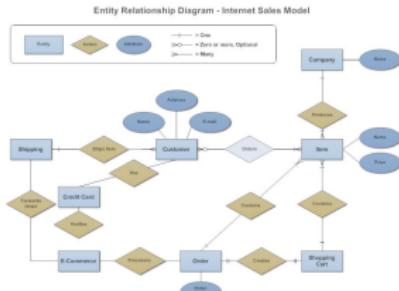
	PlaylistId	TrackId
10	1	2149
11	1	2663
12	1	2406
13	3	2920
14	3	3177
15	5	2663
16	5	350
17	5	2406
18	5	2149
19	5	864
20	5	3434
21	8	93
22	8	864
23	8	1121

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.

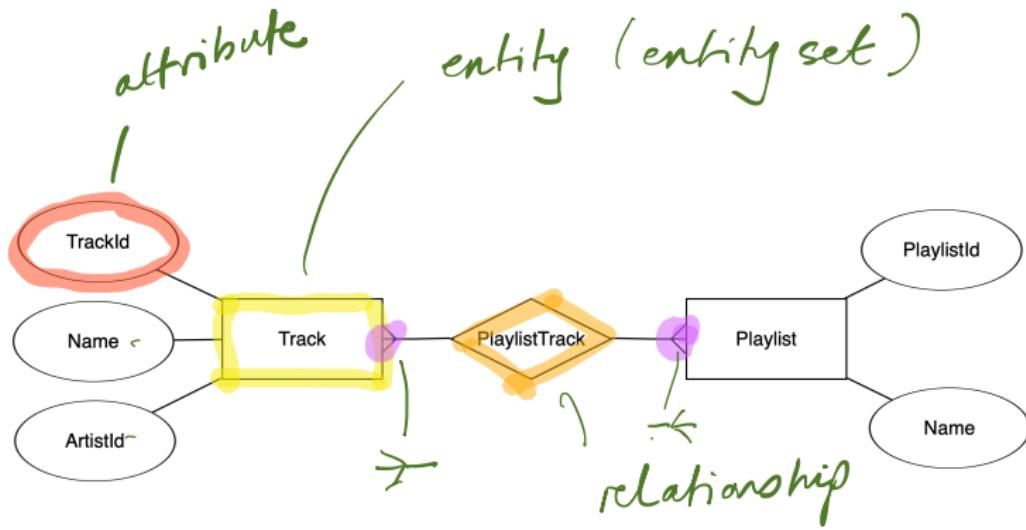
The website [erdplus.com](http://erdplus.com) provides a free online tool that allows you to create a *particular style* of Entity-Relationship Diagram.

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 Suchá	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

---

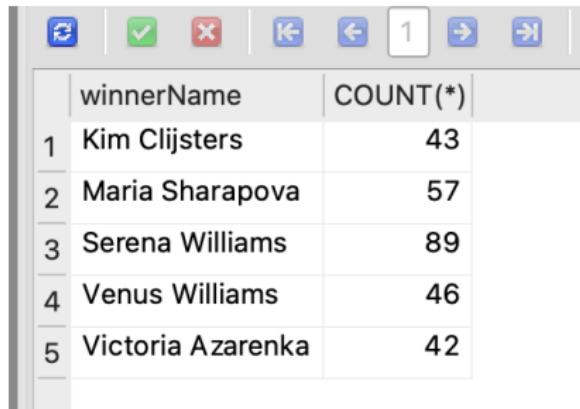
- ▶ The `SELECT` statement says to extract
  - ▶ `winnerName` from each group
  - ▶ The *row count* for each group

	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. Next to these is a page number input field containing '1'. To the right of the input field are three more arrows: a blue double-left arrow, a blue double-right arrow, and a blue double-up arrow. Below this toolbar is a table with two columns: 'winnerName' and 'COUNT(\*)'. The table has 5 rows, each with a row number from 1 to 5. 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 results are ordered by "numWins" in descending order, as specified in the query.
- Table Headers:** The table has two header cells: "winnerName" and "numWins".
- Data Rows:** There are five rows of data:

	winnerName	numWins
1	Serena Williams	89
2	Maria Sharapova	57
3	Venus Williams	46
4	Kim Clijsters	43
5	Victoria Azarenka	42
- Total Rows:** The message "Total rows loaded: 5" is displayed below the grid.

## 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



THE UNIVERSITY OF  
**WESTERN**  
AUSTRALIA

## 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	3276207
2	131	Melbourne	2865329
3	132	Brisbane	1291117
4	133	Perth	1096829
5	134	Adelaide	978100

- 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.

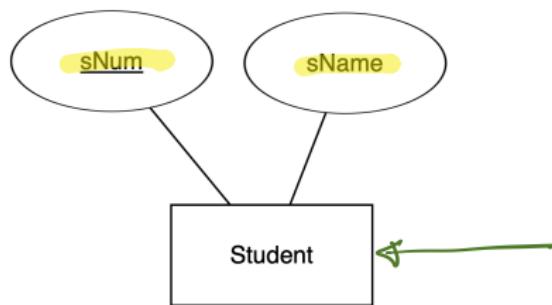
For example, the collection of *students* and the collection of *units* are each entity sets.

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");
```

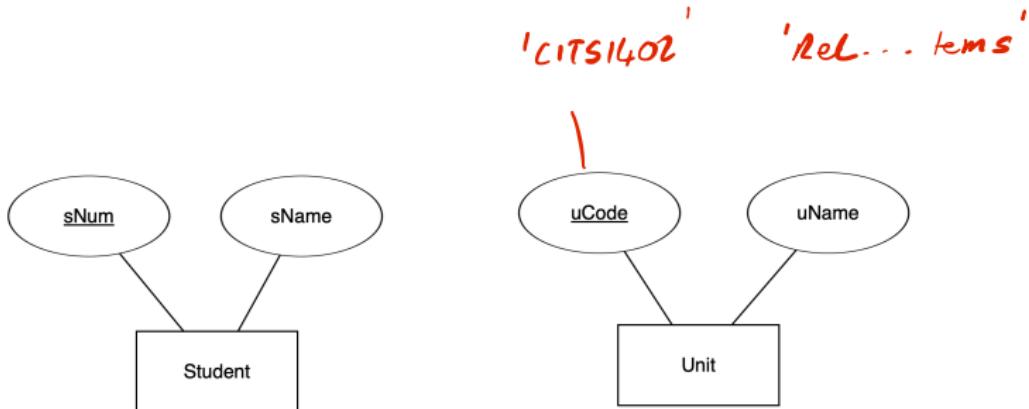
The second query, `8 INSERT INTO Student VALUES(1, "Bill");`, has the value `1` highlighted and circled in red. The status bar at the bottom of the window displays the following log entries:

- [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:

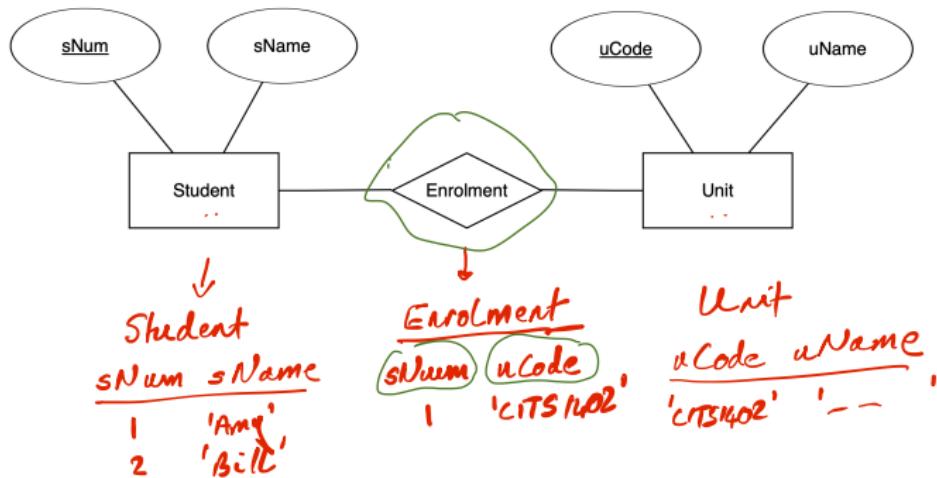
- ▶ One student
- ▶ One unit

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



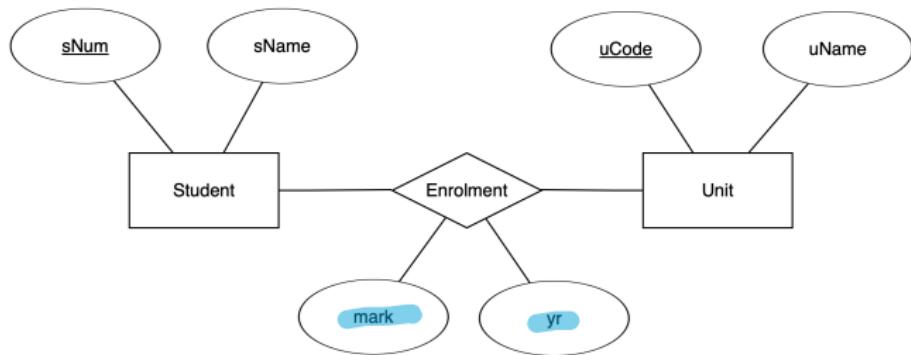
*Amy took Databases in 2018 and got a mark of 45%*

*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

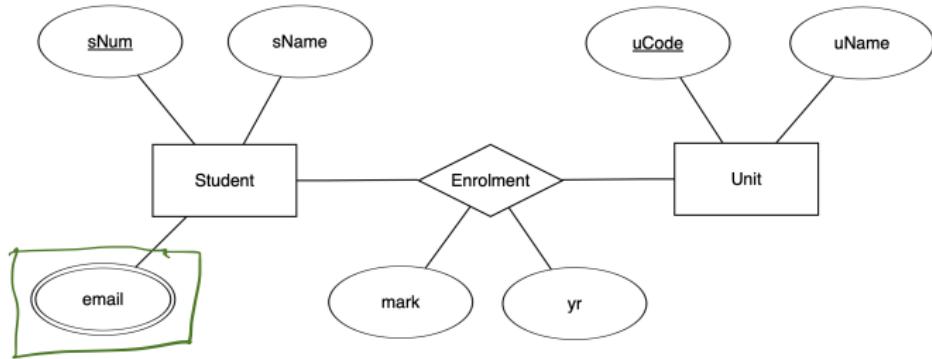
---

```
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



# 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

---

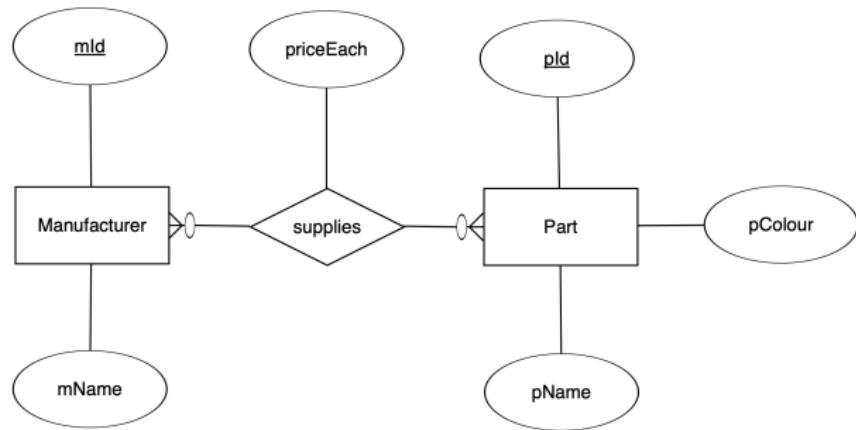
We will use a schema regarding suppliers of products.

```
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 right side groups the last four rows of table R, labeled '6.'.

	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 arrow points to the word 'INTERSECT'.

	A	B
1	3	3

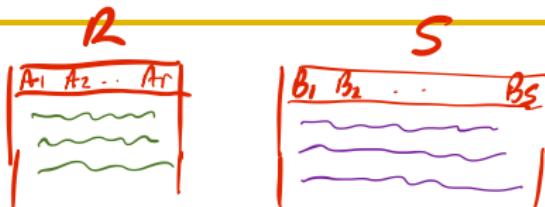
```
1 SELECT * FROM R  
2 EXCEPT  
3 SELECT * FROM S;
```

The screenshot shows the result of an EXCEPT query. It contains two rows: (1, 0) from table R and (2, 1) from table R. A red arrow points to the word 'EXCEPT'.

	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
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
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

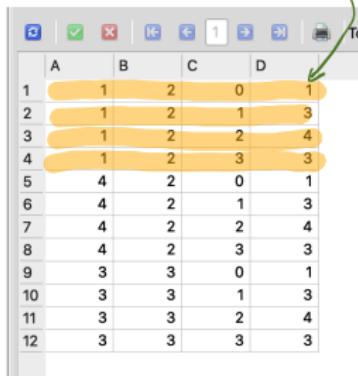
$$\underline{\sigma}_c(R \times S).$$

In other words a *join* is the result of selecting certain rows from the Cartesian product.

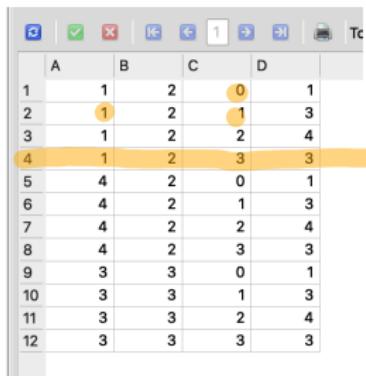
# Examples

$A = 1 \quad (\wedge)$   $C = 3$   
AND

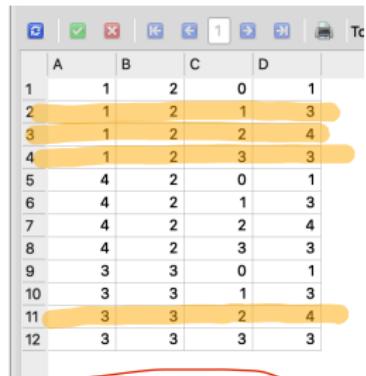
$R \bowtie S$   
 $A=1$



	A	B	C	D	Tc
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	Tc
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	Tc
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) and (2, 3). Table T has columns A and D, with data (1, 3) and (3, 3). 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	3

↑ 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 and the results grid to the `productCode` column header in the 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
-

# CITS1402

## Relational Database Management Systems

### Video 21 — Functional Dependencies I

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Redundancy

---

*Redundancy* in a DBMS refers to the storage of the same piece of data in multiple places.

While *controlled redundancy* (for example, system backups) are necessary, dealing with *uncontrolled redundancy* is a major issue in any database management system.

The concepts of *functional dependencies* and the associated theory of *normalization* is a mathematical theory dealing with redundancy.

## Redundancy

---

One of the main reasons for using relational tables for data is to avoid the problems caused by *redundant storage* of data.

For example, consider the sort of general information that is stored about a student:

- ▶ Student Number
- ▶ Name
- ▶ Address
- ▶ Date of Birth

Different parts of the university may keep different additional items of data regarding students, such as grades, financial information and so on.

# Repeating Data

---

Suppose that marks are kept in the following format:

Student Number	Name	Unit Code	Mark
14058428	John Smith	CITS1402	72
14058428	John Smith	CITS1401	68
14058428	John Smith	CITS2200	68
15712381	Jill Tan	CITS1401	88
15712381	Jill Tan	CITS1402	82

Then this table contains *redundant data*, because the student's name is repeated in numerous different rows.

## Problems with redundancy

---

Redundancy can lead to more significant problems:

- ▶ *Update Anomalies*

An *update* must be performed on *every copy* of the data item.

- ▶ *Insertion Anomalies*

A new data item (e.g. a new mark) cannot be entered without knowing *unrelated information*.

- ▶ *Deletion Anomalies*

Deletion may accidentally lose other data.

## Deletion Anomalies

---

A *deletion anomaly* occurs when a table storing redundant information becomes a proxy for storing that information properly.

For example, suppose that a company pays fixed hourly rates according to the level of an employee:

Id	Name	Level	Rate
1	Smith	10	55.00
2	Jones	8	30.00
3	Tan	10	55.00
4	White	9	42.00
⋮	⋮	⋮	⋮

This table contains the *association* between the level of an employee and the rate for that level.

## What if Jones leaves?

---

If Jones is the *only* employee at level 8 and he or she leaves, then the pay rate for Level 8s is lost.

Instead, keep a *separate table* that relates levels and rates.

Level	Rate
:	:
8	30.00
9	42.00
10	55.00
:	:

Id	Name	Level
1	Smith	10
2	Jones	8
3	Tan	10
4	White	9
:	:	

# Separating the student tables

---

Creating a separate table with *just* the basic student information.

Student Number	Name
14058428	John Smith
15712381	Jill Tan

and then the marks in a separate table.

Student Number	Unit Code	Mark
14058428	CITS1402	72
14058428	CITS1401	68
14058428	CITS2200	68
15712381	CITS1401	88
15712381	CITS1402	68

## Decomposition (Example 1)

---

In the “Jones leaves” example, the single table with columns

(Name, Level, Rate)

is replaced by *two tables* with columns

(Name, Level)

(Level, Rate)

This is called *decomposition*.

## Decomposition (Example 2)

---

In the marks example the table

(Student Number, Name, Unit Code, Mark)

is replaced by

(Student Number, Name)

(Student Number, Unit Code, Mark)

This is called *decomposition*.

# Decomposition

---

A *decomposition* of a relation schema  $R$  is a collection of two (or more) relation schemas

$$R_1, R_2, \dots, R_k$$

such that:

- ▶ The columns of each  $R_i$  are a subset of the columns of  $R$

$$R_i \subseteq R$$

- ▶ Every column of  $R$  is a column of at least one of the  $R_i$

$$R = \bigcup_{i=1}^{i=k} R_i$$

## Schema not instance

---

The ideas of *decomposition* and *normalization* relate to the *structure* of the tables and not the *contents* of the tables.

Decisions about the schema and normalization are part of the *database design* process.

## Example

---

To minimise notation, we express a schema such as

$R(\text{Student Number}, \text{Name}, \text{Unit Code}, \text{Mark})$

as

$$R = \text{SNUM}$$

( $S = \text{Student Number}$ ,  $N = \text{Name}$ ,  $U = \text{Unit Code}$ ,  $M = \text{Mark}$ ).

Then the decomposition suggested above would decompose  $R$  into

$$R_1 = \text{SN} \quad R_2 = \text{SUM}$$

## To decompose or not?

---

The fundamental question:

Is it better to use *one relation*  $R$  with attributes SNUM or *two relations*  $R_1$  and  $R_2$  with attributes SN and SUM?

More precisely, what are the trade-offs between these choices?

# Which is better?

---

If we replace  $R$  by  $R_1$  and  $R_2$ , where would the *data* go?

Student Number	Name	Unit Code	Mark
14058428	John Smith	CITS1402	72
14058428	John Smith	CITS1401	68
14058428	John Smith	CITS2200	68
15712381	Jill Tan	CITS1401	88
15712381	Jill Tan	CITS1402	82

Student Number	Name
14058428	John Smith
14058428	John Smith
14058428	John Smith
15712381	Jill Tan
15712381	Jill Tan

Student Number	Name
14058428	John Smith
14058428	John Smith
14058428	John Smith
15712381	Jill Tan
15712381	Jill Tan

# Projections

---

A moment's thought tells us that the *only possible* thing that makes sense is that

- ▶  $R_1$  must be the *projection* of  $R$  onto the attributes of  $R_1$
- ▶  $R_2$  must be the *projection* of  $R$  onto the attributes of  $R_2$

So in our example

$$R_1 = \pi_{SN}(R)$$

$$R_2 = \pi_{SUM}(R)$$

## The first question

---

In order for the decomposition to make any sense, we cannot *lose information* by storing data in  $R_1$  and  $R_2$ .

Given any *relation instance* for  $R$ , we must be able to *reconstruct*  $R$  from the corresponding relation instances of  $R_1$  and  $R_2$ .

Moreover, this must be true for *any legal data* stored in  $R$ .

## Just one example

---

We illustrate this for *a single instance*.

```
DROP TABLE IF EXISTS R;
CREATE TABLE R (S INT, N TEXT, U TEXT, M INT);

INSERT INTO R VALUES(14058428,"John Smith","CITS1401",72);
INSERT INTO R VALUES(14058428,"John Smith","CITS1402",68);
INSERT INTO R VALUES(14058428,"John Smith","CITS2200",68);
INSERT INTO R VALUES(15712381,"Jill Tan","CITS1401",88);
INSERT INTO R VALUES(15712381,"Jill Tan","CITS1402",68);
```

# Decomposing

---

Create  $R_1$  and  $R_2$  as *projections* of  $R$ :

```
CREATE TABLE R1 (S INT, N TEXT);
CREATE TABLE R2 (S INT, U TEXT, M INT);
```

```
INSERT INTO R1
    SELECT DISTINCT S, N
    FROM R;
```

```
INSERT INTO R2
    SELECT DISTINCT S, U, M
    FROM R;
```

# What do these contain?

---

```
21  
22 SELECT * FROM R1;
```

A screenshot of a database query results window. The window has a toolbar at the top with icons for refresh, checkmark, delete, and navigation. Below the toolbar is a table with two rows of data. The table has three columns: S, N, and an empty column. Row 1 contains S=14058428 and N=John Smith. Row 2 contains S=15712381 and N=Jill Tan.

	S	N	
1	14058428	John Smith	
2	15712381	Jill Tan	

```
21  
22 SELECT * FROM R2;
```

A screenshot of a database query results window. The window has a toolbar at the top with icons for refresh, checkmark, delete, and navigation. Below the toolbar is a table with five rows of data. The table has four columns: S, U, M, and an empty column. Rows 1 and 2 have S=14058428, U=CITS1401, and M=72. Rows 3 and 4 have S=14058428, U=CITS1402, and M=68. Row 5 has S=15712381, U=CITS1401, and M=88.

	S	U	M	
1	14058428	CITS1401	72	
2	14058428	CITS1402	68	
3	14058428	CITS2200	68	
4	15712381	CITS1401	88	
5	15712381	CITS1402	68	

# Recovering data

---

The screenshot shows the SQLiteStudio interface. At the top, there is a code editor window containing the following SQL query:

```
1 SELECT *
2 FROM R1 NATURAL JOIN R2;
```

Below the code editor is a toolbar with various icons for database management. To the right of the toolbar are two tabs: "Grid view" (which is selected) and "Form". Underneath the toolbar, a message says "Total rows loaded: 5". The main area displays a grid of data with the following columns: S, N, S:1, U, and M. The data is as follows:

	S	N	S:1	U	M	
1	14058428	John Smith	14058428	CITS1401	72	
2	14058428	John Smith	14058428	CITS1402	68	
3	14058428	John Smith	14058428	CITS2200	68	
4	15712381	Jill Tan	15712381	CITS1401	88	
5	15712381	Jill Tan	15712381	CITS1402	68	

Note that SQLiteStudio displays a “phantom copy” of the field *S*.

## A minimum requirement

---

A *minimum requirement* for any proposed decomposition is that

$$R = R_1 \bowtie R_2$$

so that using the two tables  $R_1$  and  $R_2$  is precisely equivalent to using  $R$ .

CITS1402  
Relational Database Management Systems

Video 22 — NULL

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## The special value `NULL`

---

Real-life databases are often incomplete, but still must be queried.

To deal with this, SQL has a special value `NULL` that means either

- ▶ The data is *absent* or *unknown*, or
- ▶ Data in this field would have no *meaning*.

The value `NULL` and the behaviour of SQL operators and functions allows the user to deal with this sensibly.

# A synthetic table

---

Query History

```
1 CREATE TABLE Scores (A INT, B INT);
2 INSERT INTO Scores VALUES (1,3), (2,4), (NULL, 4), (3, 7), (5,NULL);
3 SELECT * FROM Scores;
```

Grid view Form view

Total rows loaded: 5

	A	B
1	1	3
2	2	4
3	NULL	4
4	3	7
5	5	NULL

# Operators

---

If part of an arithmetic expression is **NULL** then so is the final value

The screenshot shows a MySQL Workbench environment. The top pane contains a SQL query:

```
3 SELECT * FROM Scores;
4
5 SELECT A, B, A+B AS C
6 FROM Scores;
```

The bottom pane displays the results of the query in a grid format:

	A	B	C	
1	1	3	4	
2	2	4	6	
3	NULL	4	NULL	
4	3	7	10	
5	5	NULL	NULL	

The results show that the addition operation  $A + B$  produces **NULL** whenever either **A** or **B** is **NULL**.

# Comparisons

---

Any normal *comparison* involving `NULL` has the value `NULL`, so neither of these queries produce any output.

```
SELECT *
FROM Scores
WHERE A == NULL;
```

```
SELECT *
FROM Scores
WHERE A != NULL;
```

So no value is *equal* to `NULL` or *not equal* to `NULL`.

## Checking for NULL

---

We can *check* to see if a “value” in the field is actually **NULL**

```
SELECT *
FROM Scores
WHERE A IS NULL;
```

The screenshot shows a database interface with the following details:

- SQL Query:

```
11
12 SELECT *
13 FROM Scores
14 WHERE A IS NULL;|
```

The last line of the query, "14 WHERE A IS NULL;", is highlighted in light blue.
- Result Set:

	A	B	
1	NULL	4	

The result set displays one row with columns A and B. Column A contains the value "NULL", and column B contains the value "4".
- Toolbar:

Contains icons for refresh, checkmark, delete, back, forward, and page number (1).

# Aggregate functions

---

Aggregate functions *skip* NULL values.

```
16  
17 SELECT COUNT(A), SUM(A), AVG(A)  
18 FROM Scores;
```

The screenshot shows a database interface with a code editor at the top containing the SQL query. Below it is a grid view showing the results of the query. The results table has three columns: COUNT(A), SUM(A), and AVG(A). The data row shows values 1, 4, and 2.75 respectively. The interface includes standard database navigation buttons (refresh, save, delete, etc.) and a status bar indicating "Total rows loaded: 1".

COUNT(A)	SUM(A)	AVG(A)
1	4	2.75

The screenshot shows a database grid with three columns labeled A, B, and C. The data rows are numbered 1 through 5. Row 1 contains values 1, 1, and 3. Row 2 contains values 2, 2, and 4. Row 3 contains values NULL, 4, and 4. Row 4 contains values 3, 7, and 7. Row 5 contains values 5, NULL, and 5. The grid has a header row and five data rows.

	A	B	C
1	1	3	
2	2	4	
3	NULL	4	
4	3	7	
5	5	NULL	

## Subqueries with IN

---

```
SELECT * FROM Scores  
WHERE A IN  
(SELECT B FROM Scores);
```

10  
17 SELECT \* FROM Scores  
18 WHERE A IN  
19 (SELECT B FROM Scores);|

The screenshot shows a MySQL Workbench environment. The top part is a code editor with the SQL query displayed. The bottom part is a results viewer showing a table with three columns: A, B, and C. The data row has values 1, 3, and 7 respectively. The 'B' column is highlighted with a light blue background.

	A	B	C
1	1	3	7

## Subqueries with NOT IN

---

```
SELECT * FROM Scores  
WHERE A NOT IN  
(SELECT B FROM Scores);
```

```
16  
17 SELECT * FROM Scores  
18 WHERE A NOT IN  
19 (SELECT B FROM Scores);
```

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

- Code Area:** Lines 16 through 19 of the SQL script are shown. Line 19 contains the subquery used in the NOT IN clause.
- Execution Results:** Below the code, there is a grid view window. The top bar of the window has tabs for "Grid view" (which is selected) and "Form".
  - Toolbar:** Includes standard database icons for refresh, insert, delete, and search.
  - Status Bar:** Shows "Total rows loaded: 0".
- Data Grid:** The main area displays a single row with columns labeled "A" and "B". Both columns are empty.

Why does this produce no output?

## Outer Joins

---

All the joins that we have seen are *inner joins*.

If table  $T$  results by joining  $T1(A, B, C)$  to  $T2(D,E,F)$  then  $T$  has schema  $T(A,B,C,D,E,F)$ .

In principle, the table  $T$  is formed when SQL

- ▶ *concatenates* a row of  $T1$  with a row of  $T2$
- ▶ *tests* whether this new “long row” meets the *join condition* in all possible ways.

If there is a row of  $T1$  that *does not meet* the join condition with *any* row of  $T2$  then it does not occur in  $T$ .

# Sample Schema

---

16  
17 SELECT \* FROM Student;

This screenshot shows a database interface with a toolbar at the top and a table below. The table has columns 'sNum' and 'sName'. The data is as follows:

	sNum	sName
1	1	Amy
2	2	Bill
3	3	Chin
4	4	Dagmar
5	5	Eng Guan

16  
17 SELECT \* FROM Enrolment;

Grid view

This screenshot shows a database interface with a toolbar at the top and a table below. The table has columns 'sNum', 'uCode', 'yr', and 'mark'. The data is as follows:

	sNum	uCode	yr	mark
1	1	MATH1001	2018	55
2	1	MATH1002	2019	68
3	2	CITS1402	2019	30
4	2	CITS1402	2020	NULL

16  
17 SELECT \* FROM Unit;

This screenshot shows a database interface with a toolbar at the top and a table below. The table has columns 'uCode' and 'uName'. The data is as follows:

	uCode	uName
1	CITS1401	Programming
2	CITS1402	Databases
3	CHEM1001	Chemistry
4	PHYS1001	Physics
5	MATH1001	Maths I
6	MATH1002	Maths II

## Normal join

---

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

```
7 SELECT *
8 FROM Student S JOIN Enrolment E
9 ON S.sNum = E.sNum;|
```

Grid view Form view

Total rows loaded: 4

	sNum	sName	sNum:1	uCode	yr	mark	
1	1	Amy		1 MATH1001	2018	55	
2	1	Amy		1 MATH1002	2019	68	
3	2	Bill		2 CITS1402	2019	30	
4	2	Bill		2 CITS1402	2020	NULL	

# Left join

```
SELECT *
FROM Student S LEFT OUTER JOIN Enrolment E
ON S.sNum = E.sNum;
```

```
17 SELECT *
18 FROM Student S LEFT OUTER JOIN Enrolment E
19 ON S.sNum = E.sNum;
20 |
```

Grid view Form view

Total rows loaded: 7

	sNum	sName	sNum:1	uCode	yr	mark	
1	1	Amy	1	MATH1001	2018	55	
2	1	Amy	1	MATH1002	2019	68	
3	2	Bill	2	CITS1402	2019	30	
4	2	Bill	2	CITS1402	2020	NULL	
5	3	Chin	NULL	NULL	NULL	NULL	
6	4	Dagmar	NULL	NULL	NULL	NULL	
7	5	Eng Guan	NULL	NULL	NULL	NULL	

## Left outer join

---

In a **LEFT OUTER JOIN** every row in the *first-named table* (the left-table) appears at least once, *even if it does not match* any right-table rows.

- ▶ Left-table row satisfies **JOIN** condition with a right-table row
  - Row appears in joined table as usual
- ▶ Left-table row does not satisfy **JOIN** condition with any right-table row
  - Row appears in joined table with “dummy right-table row” consisting entirely of **NULLs**

Why is this useful?

## Adding zeros to counts

---

```
SELECT sName, COUNT(uCode)
FROM Student S LEFT OUTER JOIN Enrolment E
ON S.sNum = E.sNum
GROUP BY S.sNum;
```

20  
21 SELECT sName, COUNT(uCode)  
22 FROM Student S LEFT OUTER JOIN Enrolment E  
23 ON S.sNum = E.sNum  
24 GROUP BY S.sNum;  
25 |

Grid view

Total rows loaded

	sName	COUNT(uCode)
1	Amy	2
2	Bill	2
3	Chin	0
4	Dagmar	0
5	Eng Guan	0

## How many staff do you supervise?

---

```
SELECT E1.employeeNumber, COUNT(E2.reportsTo)
FROM employees E1 JOIN employees E2
ON E2.reportsTo = E1.employeeNumber
GROUP BY E1.employeeNumber;
```

```
SELECT E1.employeeNumber, COUNT(E2.reportsTo)
FROM employees E1 LEFT OUTER JOIN employees E2
ON E2.reportsTo = E1.employeeNumber
GROUP BY E1.employeeNumber;
```

## Outer joins

---

Some final remarks,

- ▶ `LEFT OUTER JOIN` is the same as `LEFT JOIN`
- ▶ `RIGHT JOIN` and `RIGHT OUTER JOIN` are valid SQL, but *not implemented* in SQLite
- ▶ `FULL JOIN` and `FULL OUTER JOIN` are also valid SQL, but also *not implemented* in SQLite.

# CITS1402

## Relational Database Management Systems

### Video 23 — Data Integrity

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



THE UNIVERSITY OF  
**WESTERN**  
AUSTRALIA

# Data Integrity

---

Databases are rarely static objects holding unchanging data.

In general, the contents are *constantly changing* as users

- ▶ Create data,
- ▶ Read data,
- ▶ Update data,
- ▶ Delete data.

It is *vital* that the database remains in a *logically consistent state* — so there are many mechanisms to maintain *data integrity*.

## Low level

---

What appears to the user to be a single operation may actually need *several* database operations to complete.

So the operation “Transfer \$100000 from Account A to Account B” actually requires two steps:

- ▶ Increase balance of Account B by \$100000
- ▶ Decrease balance of Account A by \$100000

Between the first and second step, the database will *temporarily* be in an inconsistent state.

# Transactions

---

*Transactions* are a low-level mechanism to ensure that

- ▶ No other database operations (or users) can manipulate the data while it is inconsistent  
*In a multi-user database (e.g. a ticketing website), several users may be pursuing the same tickets at the same time.*
- ▶ The system can recover from unexpected external events  
*Power cuts, disk crashes, internet drop-outs, etc*

We will study transactions in more detail later, but as SQLite is not a true multi-user database, it is less important than for other RDBMS.

# Basic data integrity

---

Declaring *keys* is a key component<sup>1</sup> of data integrity.

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

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

---

<sup>1</sup>sorry for the pun

## If fields must have values

---

Sometimes it is sensible for a field to take the value `NULL`, but other times it isn't.

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

```
CREATE TABLE Unit (
    uCode TEXT PRIMARY KEY,
    uName TEXT NOT NULL );
```

Students and Units must have names.

# Referential Integrity

---

In an RDBMS, values in one table often *refer to* tuples in another.

*Referential integrity* means that these references should always refer to a legitimate tuple in the second table.

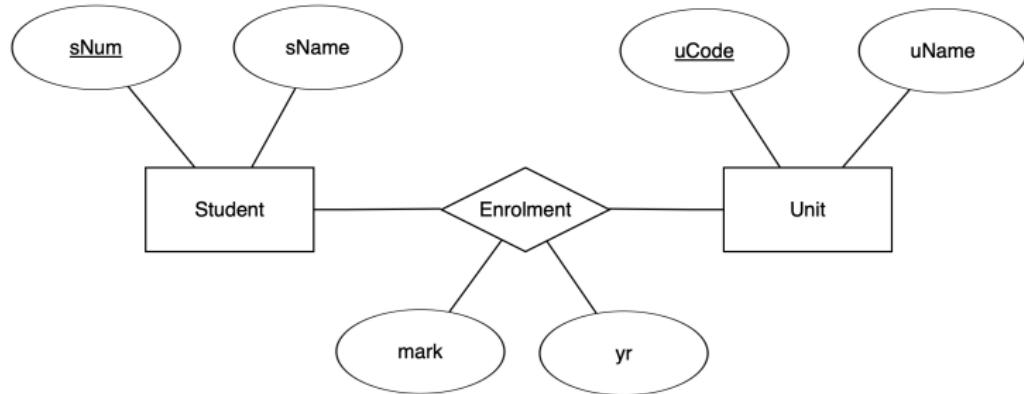
If Table A has a column whose values refer to Table B, then

- ▶ Table A is the *child table*
- ▶ Table B is the *parent table*

“Children point to parents”

# The university schema

---



## The child table

---

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

# Current contents

---

23 SELECT \* FROM Student;|

The screenshot shows a database interface with a toolbar at the top containing icons for refresh, insert, delete, update, and search. Below the toolbar is a table with two columns: sNum and sName. The data consists of five rows with values 1 through 5 respectively, followed by the names Amy, Bill, Chin, Dagmar, and Eng Guan.

sNum	sName
1	Amy
2	Bill
3	Chin
4	Dagmar
5	Eng Guan

23 SELECT \* FROM Enrolment;|

The screenshot shows a database interface with a toolbar at the top containing icons for refresh, insert, delete, update, and search. Below the toolbar is a table with four columns: sNum, uCode, yr, and mark. The data consists of four rows with values 1 through 4 respectively, followed by the corresponding uCode (MATH1001, MATH1002, CITS1402, CITS1402), years (2018, 2019, 2019, 2020), and marks (55, 68, 30, NULL).

sNum	uCode	yr	mark
1	MATH1001	2018	55
2	MATH1002	2019	68
3	CITS1402	2019	30
4	CITS1402	2020	NULL

23 SELECT \* FROM Unit;|

The screenshot shows a database interface with a toolbar at the top containing icons for refresh, insert, delete, update, and search. Below the toolbar is a table with two columns: uCode and uName. The data consists of six rows with values 1 through 6 respectively, followed by the unit codes CITS1401, CITS1402, CHEM1001, PHYS1001, MATH1001, and MATH1002, and their corresponding names Programming, Databases, Chemistry, Physics, Maths I, and Maths II.

	uCode	uName
1	CITS1401	Programming
2	CITS1402	Databases
3	CHEM1001	Chemistry
4	PHYS1001	Physics
5	MATH1001	Maths I
6	MATH1002	Maths II

## A constraint

---

The business logic says that at all times,

- ▶ The values in `Enrolment.sNum` should also be in `Student.sNum`
- ▶ The values in `Enrolment.uCode` should also be in `Unit.uCode`

In particular, whenever a new tuple is entered into `Enrolment`

```
INSERT INTO Enrolment VALUES(5, 'MATH1002', 2020, 66);
```

the system should check that there actually *is* a student with that number and a unit with that code.

## Prepare the child table

---

```
CREATE TABLE Enrolment (
    sNum INTEGER,
    uCode TEXT,
    yr INTEGER,
    mark INTEGER,
    FOREIGN KEY(sNum) REFERENCES Student(sNum),
    FOREIGN KEY(uCode) REFERENCES Unit(uCode)
);
```

# Enforcing key constraints

---

Obviously bad rows are simply *not allowed*.

```
INSERT INTO Enrolment VALUES(6, 'MATH1003', 2020, 66);
```

```
1 pragma foreign_keys;  
2  
3 INSERT INTO Enrolment VALUES(6, 'MATH1003', 2020, 66);|
```

Status

ⓘ [12:06:15] Query finished in 0.000 second(s).

❗ [12:07:30] Error while executing SQL query on database 'marks': FOREIGN KEY constraint failed

## Updates

---

Suppose I want to *update* student numbers to a 2-digit code, so (1, Amy) becomes (11, Amy) and so on.

```
UPDATE Student  
SET sNum = sNum + 10;
```

If this happened, then the values in `Enrolment.sNum` would be *dangling pointers*.

So the foreign key constraint *prevents* the operation.

## Alternative options

---

What are some *other behaviours* that make sense when updating or deleting rows in these tables?

- ▶ *Forbidding* the operation

This is the default, and operations that would cause a referential integrity violation are not executed.

- ▶ *Cascading* the operation

This means to “fix up” the referential integrity violation by doing the same thing to the row in the child table as has happened to the matching row in the parent table.

- ▶ Using **NULL** to allow the operation to occur, but leaving the database in a state that is “controllably inconsistent”.

## Specifying a cascade

---

```
CREATE TABLE Enrolment (
    sNum INTEGER,
    uCode TEXT,
    yr INTEGER,
    mark INTEGER,
    FOREIGN KEY(sNum) REFERENCES Student(sNum)
        ON UPDATE CASCADE
        ON DELETE SET NULL,
    FOREIGN KEY(uCode) REFERENCES Unit(uCode)
);
```

# Cascading updates

---

```
UPDATE Student  
SET sNum = sNum + 10;
```

The change *cascades* down to the child table.

Query

```
1 UPDATE Student SET sNum = sNum + 10;  
2  
3 SELECT * FROM Enrolment;  
4 |
```

Grid view

	sNum	uCode	yr	mark	
1	11	MATH1001	2018	55	
2	11	MATH1002	2019	68	
3	12	CITS1402	2019	30	
4	12	CITS1402	2020	NULL	

## The options

---

In SQLite, the options to follow `ON UPDATE` and `ON DELETE` include:

- ▶ `RESTRICT` or `NO ACTION`

This *prevents* the update/delete.

- ▶ `CASCADE`

This *cascades* the update.

- ▶ `SET NULL`

Replaces the affected values in the child-table by `NULL`

## A sample question - what is the output?

---

```
CREATE TABLE R (A INT PRIMARY KEY);
CREATE TABLE S (B INT PRIMARY KEY,
    FOREIGN KEY (B) REFERENCES R(A) ON UPDATE CASCADE);
CREATE TABLE T (C INT PRIMARY KEY,
    FOREIGN KEY (C) REFERENCES S(B) ON UPDATE CASCADE);
```

If  $R = \{1, 2, 3, 4, 5, 6\}$ ,  $S = \{1, 2, 4, 6\}$  and  $T = \{1, 2, 6\}$  then  
what is the result of

```
UPDATE R
SET A = A + 10
WHERE A < 5;
```

```
SELECT SUM(C)
FROM T;
```

CITS1402  
Relational Database Management Systems

Video 24 — Entity Relationship Diagrams III

GORDON ROYLE

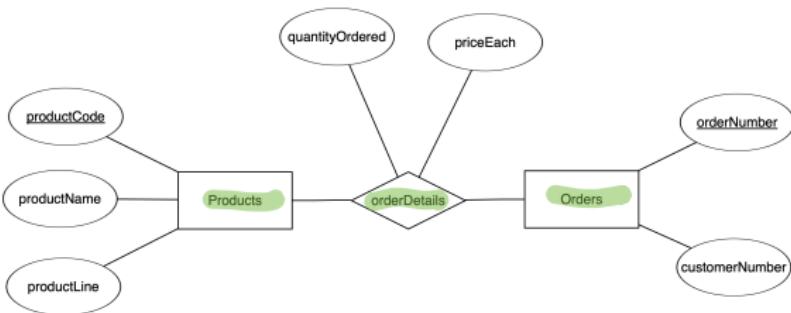
DEPARTMENT OF MATHEMATICS & STATISTICS



# ERD

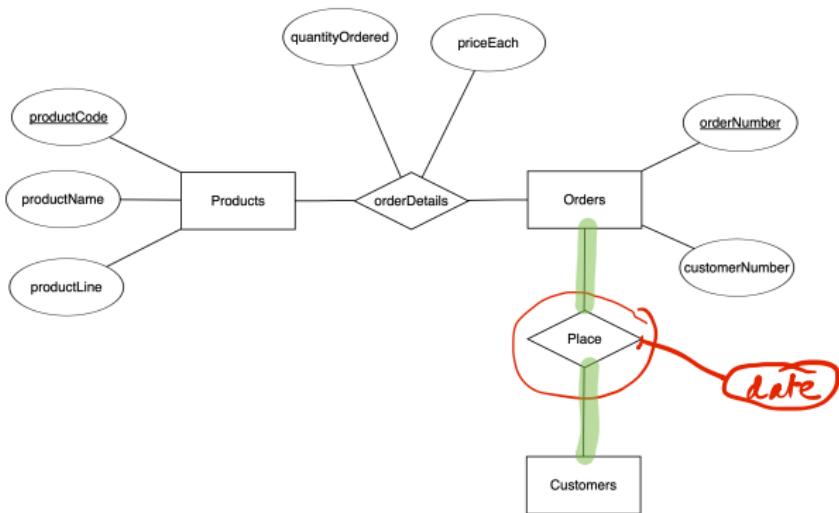
---

We'll finish the basic features entity-relationship diagrams, and look briefly at [ERDPlus.com](http://ERDPlus.com).



# Classic Models

---



The relationship is that “customers *place* orders”.

# Relationships

---

How *relationships* are implemented in the database depends on how the business logic *constraints* the relationship.

These constraints are usually called *cardinality constraints* of the relationship, and are included in an ERD through specific symbols.

*"crow's foot  
notation"*

## An example

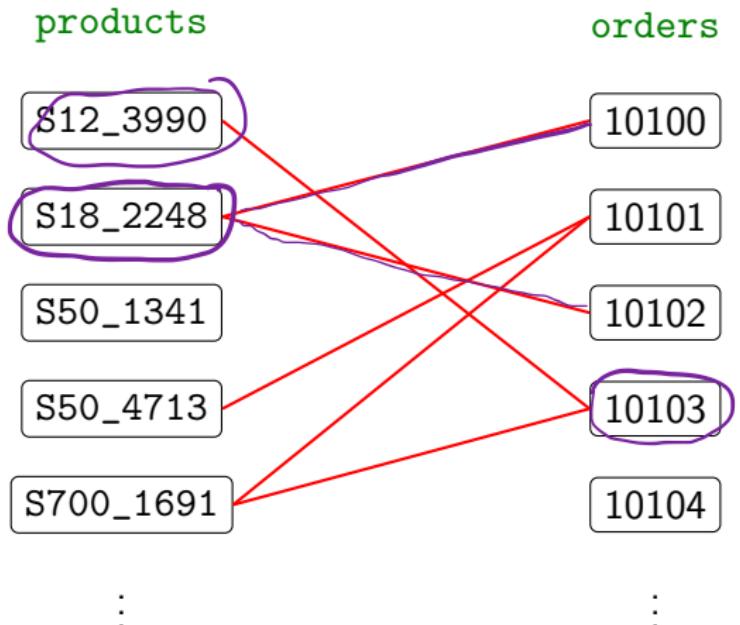
---

In ClassicModels, the table `orderdetails` records connections between `products` and `orders`.

products	orders
S12_3990	10100
S18_2248	10101
S50_1341	10102
S50_4713	10103
S700_1691	10104
⋮	⋮

# The relationship

---



# Cardinality

---

Cardinality is a fancy word for the “size” or “number” of something.

*Restrictions* on *how many times* an element (of either entity set) can occur in a relation are called *cardinality constraints*.

(S700\_1691, 10101)  
(S700\_1691, 10103)

- ▶ One product can appear in *several* orders. ✓
- ▶ One order can contain *several* products. ✓

## Zero, one or many

---

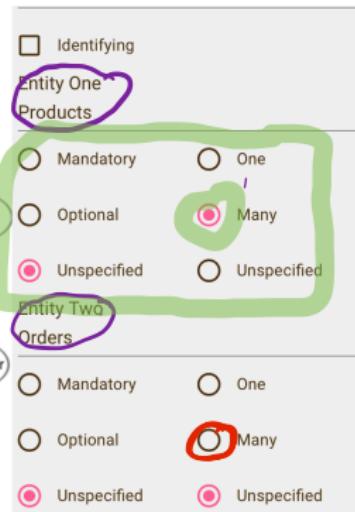
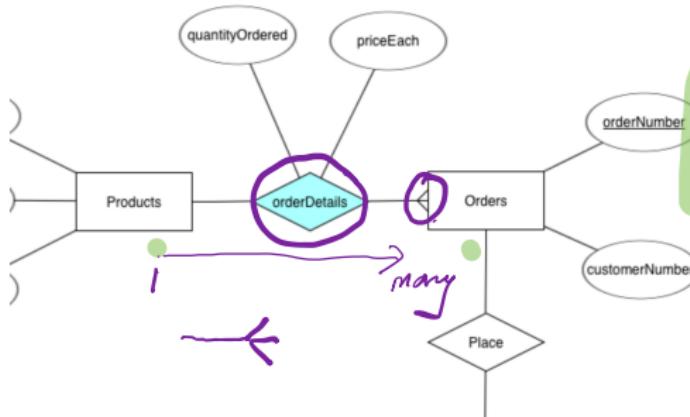
We are not concerned with *exact* limits, but just

- ▶ Zero – Is it possible for an entity to be in zero tuples?
- ▶ One – Is it possible for an entity to be in one tuple?
- ▶ Many – Is it possible for an entity to be in many tuples?

These are *decided* by business rules, and then *recorded* in the ERD.

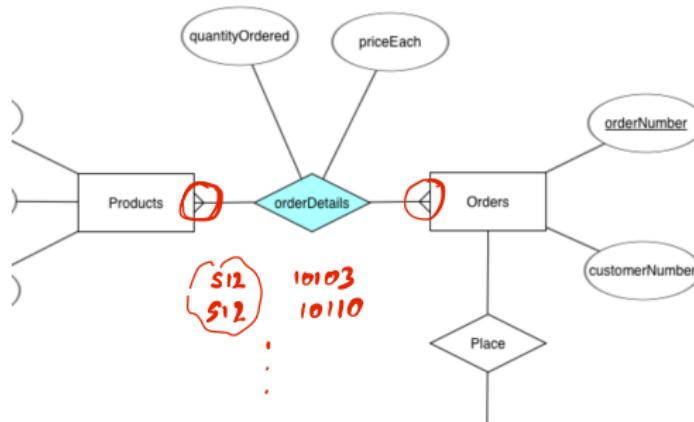
# How to record them?

One product can appear in *many* orders.



# The other way round

One order can contain *many* products.



Identifying	
Entity One	Products
<input type="radio"/> Mandatory	One
<input type="radio"/> Optional	Many
<input checked="" type="radio"/> Unspecified	Unspecified
Entity Two	
Orders	
<input type="radio"/> Mandatory	One
<input type="radio"/> Optional	Many
<input checked="" type="radio"/> Unspecified	Unspecified

## Participation constraints

---

Does it make sense for there to be an **order** with no products?

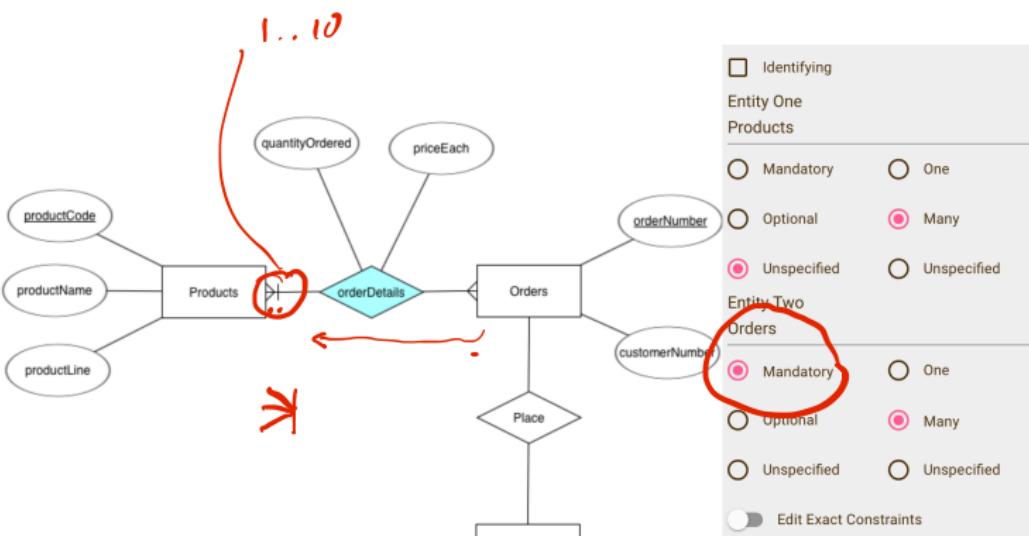
If not, then **every order** must appear in the relation **orderdetails**.

So there is **mandatory participation** of **orders** in the relation.

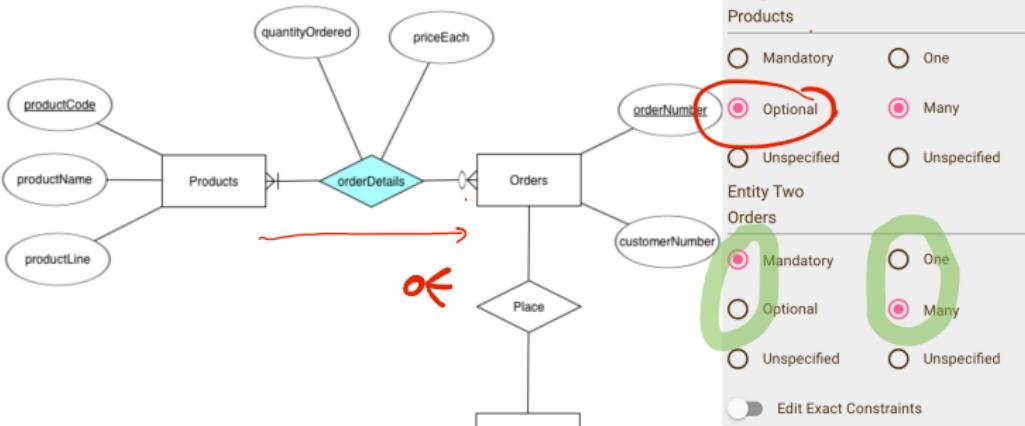
Classic Models decides that it is reasonable for **a product** to be in the database even if it has not yet been ordered.

So there is **optional participation** of **products** in the relation

# Mandatory participation

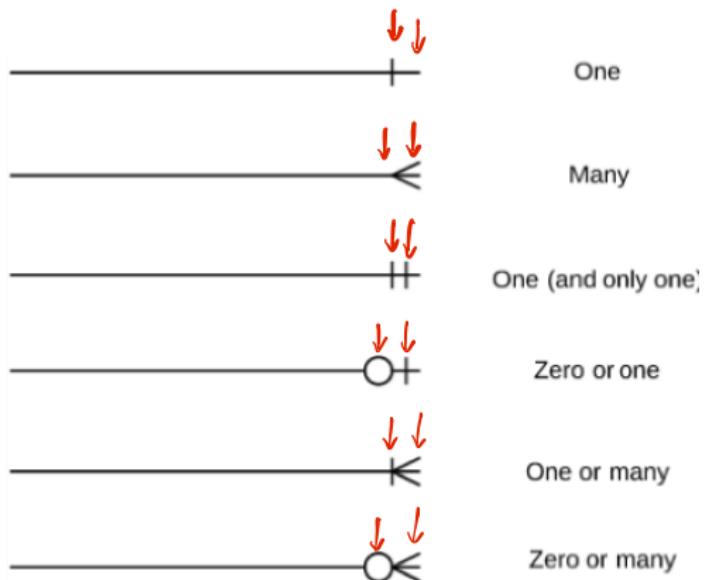


# Optional participation



# Crow's foot symbols

---



# Minimum and Maximum

---

To determine the cardinalities we fill in the blanks in the sentence with *zero*, *one* or *many*.

A *single* order contains

- ▶ *at least*  products
- ▶ *at most*  products

10103

or *as details*

A *single* product is contained in

- ▶ *at least*  orders
- ▶ *at most*  orders

10103 512-1799  
10103 . 512-1920

10101 . 512-1920

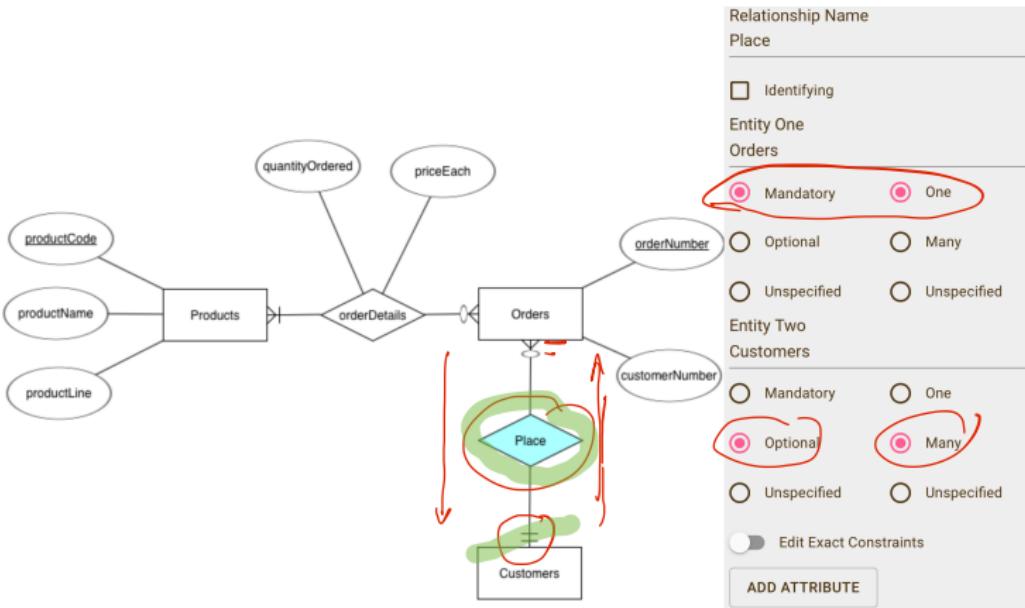
## What about placing orders?

---

What is the nature of the relationship between customers and orders?

- ▶ A customer can place zero, one or many orders
- ▶ An order must be placed by *one and only one* customer

# One and only one



## Implementation

---

A relationship that is *many-to-many* must be implemented as a separate table in the database.

But a relationship like *Place* does not need its own table — it is sufficient to have an *attribute* in *Orders* that refers to a *unique* customer.

This can be efficiently implemented as a foreign key to the *Customers* table.

structure

Table name: **orders**  WITHOUT ROWID

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	Default
1	orderNumber	int (11)	key						NULL
2	orderDate	date							NULL
3	requiredDate	date							NULL
4	shippedDate	date							NULL
5	status	varchar (15)							NULL
6	comments	text							NULL
7	customerNumber	int (11)							NULL

↑ Foreign key constraint for customerNumber

↓ Primary key constraint for orderNumber

Type Name Details

1 PRIMARY KEY	(orderNumber)	
2 FOREIGN KEY	orders_jbfk_1 (customerNumber) REFERENCES customers (customerNumber)	

# CITS1402

## Relational Database Management Systems

### Video 25 — Data Manipulation Language (DML)

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Changing data

---

So far we have focussed on querying a *pre-existing* database.

In practice, the data changes as the database is used.

- ▶ New rows are *inserted*
- ▶ Existing rows are *updated*
- ▶ Obsolete rows are *deleted*

The commands for these operations are collectively known as the Data Manipulation Language, or DML.

# The DML

---

The main DML commands in SQL/SQLite are

- ▶ `INSERT`
- ▶ `UPDATE`
- ▶ `DELETE`

We've already used `INSERT` to insert one or more rows.

```
INSERT INTO Student VALUES (6, 'Derek');
```

```
INSERT INTO Student VALUES (6, 'Derek'), (7, 'Harmony');
```

# Updating a row

---

To *update* values in certain rows:

```
UPDATE <table_name>
SET <assignments_to_columns>
WHERE <bool-conditions> ;
```

For example

```
UPDATE Employees
SET extension = 'x5580'
WHERE employeeNumber = 1002;
```

# Analysis

---

The table to be updated is Employees.

Each row in turn is examined and the specified *action*:

```
SET extension = 'x5580'
```

is applied to those rows that satisfy the *condition*

```
WHERE employeeNumber = 1002
```

## Scaling marks

---

The **SET** part of the statement can refer to values in the column being changed, or other columns.

```
UPDATE Enrolment  
SET mark = mark + 5  
WHERE uCode = 'CITS1402'  
AND yr = 2019;
```

## An empty condition is true

---

With no `WHERE` condition, the action happens to *every row*.

```
UPDATE Enrolment  
SET mark = mark + 5;
```

This adds five points to *every mark*.

## More than one column

---

The **SET** statement can alter more than one column.

```
UPDATE Student
SET sName = 'Amy Chan',
    email = 'achan02@uwa.edu.au'
WHERE sNum = 1;
```

## Deleting rows

---

Deleting rows follows the same structure:

```
DELETE FROM <tablename>
WHERE <bool-conditions> ;
```

Each row that satisfies the condition is *deleted* from the table.

## Remove failed students

---

```
DELETE FROM Enrolment  
WHERE mark < 50;
```

What does this do to the current students who have yr=2022 and  
mark=NULL?

## Don't forget the condition

---

Don't forget the **WHERE** condition when using **DELETE**.

```
DELETE FROM Student;
```

What does this do?

# Subqueries in DML

---

DML statements can use subqueries in various ways:

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

```
INSERT INTO topStudents
SELECT sNum, yr, uCode
FROM Enrolment WHERE
    (yr, uCode, mark) IN (SELECT yr, uCode, MAX(mark)
                           FROM Enrolment
                           GROUP BY yr, uCode);
```

# CITS1402

## Relational Database Management Systems

Video 27 — Views

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Frequently-used data

---

Suppose that a ClassicModels employee frequently needs to use data in the following form:

	ordernumber	customername	customernumber	ordervalue
1	10100	Online Diecast Creations Co.	363	10223.83
2	10101	Blauer See Auto,Co.	128	10549.01
3	10102	Vitachrome Inc.	181	5494.78
4	10103	Baane Mini Imports	121	50218.95
5	10104	Euro+ Shopping Channel	141	40206.2
6	10105	Danish Wholesale Imports	145	53959.21
7	10106	Rovelli Gifts	278	52151.81
8	10107	Land of Toys Inc.	131	22292.62
-	10108	Crail & Sons Co.	295	51001.22

This lists the *order number*, the *name* and *customer number* of the customer who made that order, and the *total value* of that order.

## Frequently-used tables

---

This involves three tables and so requires two joins:

```
SELECT orderNumber,
       customerName,
       customerNumber,
       ROUND(SUM(priceEach * quantityOrdered), 2) AS
           orderValue
  FROM customers
    JOIN orders USING (customerNumber)
    JOIN orderDetails USING (orderNumber)
 GROUP BY orderNumber;
```

## Cumbersome joins

---

The Classic Models employee needs to use this information in her *further queries* such as the following

```
SELECT customerName, SUM(orderValue) FROM
(SELECT orderNumber,
customerName,
customerNumber,
ROUND(SUM(priceEach * quantityOrdered),2) AS
orderValue
FROM customers
JOIN orders USING (customerNumber)
JOIN orderdetails USING (orderNumber)
GROUP BY orderNumber)
GROUP BY customerNumber;
```

## If only ...

---

This is very tedious, so she wishes that there was an *actual table* in the database that contained just this information.

```
orderSummary(customerName TEXT,  
             customerNumber INTEGER,  
             orderNumber INTEGER,  
             orderValue REAL)
```

But we *don't* want to store information in two places, which introduces potential anomalies.

## Views to the rescue

---

A *view* is a “virtual table” defined using a *stored query*.

```
CREATE VIEW orderSummary AS
SELECT orderNumber,
       customerName,
       customerNumber,
       SUM(priceEach * quantityOrdered) AS orderValue
  FROM customers
    JOIN orders USING (customerNumber)
    JOIN orderDetails USING (orderNumber)
 GROUP BY orderNumber;
```

## Enjoying the view

---

The *view* is called `orderSummary` and its schema is inherited from the data types of the columns in the query.

As far as the user is concerned, `orderSummary` behaves just like any other table.

```
SELECT customerName, SUM(orderValue)
FROM orderSummary
GROUP BY customerNumber;
```

But what happens “behind the scenes” is that SQLite *re-runs* the query that defines the view, and uses the *newly-computed values*.

## What did we gain?

---

For **Classic Models**, the fact that an order comprises several `orderdetails` makes it awkward to deal with *individual orders*.

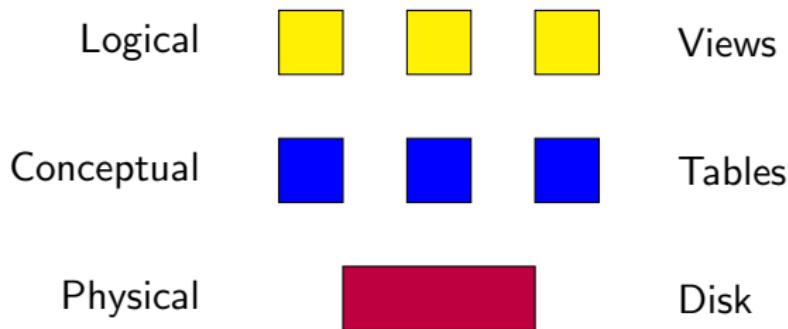
The database has the wrong *granularity* for the application — `orderdetails` is too *fine-grained*.

The view provides an *interface* to the end-user that *hides the unwanted detail*, without needing to *change* the database at all.

## Jennifer on views

---

Database guru *Jennifer Widom* describes this as a hierarchy



<https://youtu.be/x81SX-zqZIc>  
<https://youtu.be/NRl-Fh1Q7fM>

## The benefits of views

---

The structure of a database is determined by many factors:

- ▶ The requirements summarised in the ERD
- ▶ Schema normalization and redundancy removal
- ▶ Security and access control
- ▶ Cost and efficiency

This can lead to a structure that is not necessarily easy to *use*, or not necessarily easy for an “end-user” to use.

## Benefits of views cont.

---

The Database Administrator (DBA) can define *customised tables* for each type of user.

A win-win situation:

- ▶ The user has tables containing *exactly* the desired information
- ▶ The DBA has an optimised database that is easy to maintain
- ▶ The DBA has fine-grained control over access to sensitive data

As with all *abstraction layers*, an additional benefit is that the underlying *implementation* can be changed, and by simply changing the views, the end-users need not change any of their queries.

CITS1402  
Relational Database Management Systems

Video 28 — Views II

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## More view terminology

---

Two important concepts:

- ▶ *Updatable* views

Updates to the tables can be performed using the *view*

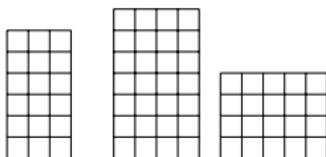
- ▶ *Materialized* views

A view is *materialized* if the results of the query it represents have actually been *stored* in the database

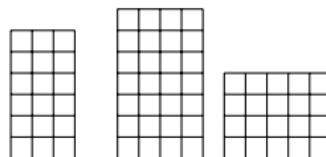
# Updatable views

---

A view is *updatable* if you can use **DELETE**, **INSERT** and **UPDATE** statements on it.



View Query



View Query



## Updating

---

The rows of the view are only *transient* — they only appear as the *result* of a query.

What does it mean to *insert*, *delete* or *update* in this context?

The system has to work out how to alter the *actual tables* so that when the view is accessed again, it has been changed accordingly.

## An updatable view

---

For a simple example, consider

```
CREATE VIEW marksOnly AS  
SELECT sNum, mark  
FROM Enrolment;
```

## Deletion

---

What should this statement do?

```
DELETE FROM marksOnly  
WHERE sNum = 1;
```

To ensure **marksOnly** has no (virtual) rows with **sNum = 1**

- ▶ Delete every row from **Enrolment** with **sNum=1**

So SQL alters the query to

```
DELETE FROM Enrolment  
WHERE sNum = 1;
```

## Not updatable

---

Consider a different view

```
CREATE VIEW studentAverage AS  
SELECT sNum, AVG(mark) as wam  
FROM Enrolment  
GROUP BY sNum;
```

There is no obvious way in which SQL can update the values in the original tables in order to produce the same effect as:

```
UPDATE studentAverage  
SET wam = 85  
WHERE sNum = 1;
```

## Materialized views

---

Suppose a view represents a *very complex query* on some data that changes infrequently, but is queried often.

In this situation, it makes sense to *store the results* of the view query in an actual database table — essentially this is what *caching* is in any computing environment.

So the “virtual” table has *materialized* as an actual table.

This is called a *materialized view*.

## Dealing with change

---

If one of the base tables *changes* due to insertion, update or deletion, then there may be a difference between

- ▶ The result of the view query (with the changed tables)
- ▶ The stored values of the materialized view

So the query and the materialized view are now *out of sync*, at least until the materialized view is *refreshed*.

## Refreshing views

---

*Refreshing* the view is the process of *re-running* the query and *saving* the new output based on the changed tables.

This can be done:

- ▶ *Automatically* (whenever the base tables change), or
- ▶ On a *schedule* (daily, weekly etc), or
- ▶ By running REFRESH MATERIALIZED VIEW (or similar)

(The term “materialized view” with no further qualification is taken to mean an auto-refresh materialized view.)

## In SQLite

---

There is a *wide variation* in the way that materialized and updatable views are implemented across different implementations of SQL.

SQLite *does not implement* materialized views or updatable views.

However, most of the functionality of materialized views can be implemented using *triggers*.

These will be the subject of a later video.

# CITS1402

## Relational Database Management Systems

### Video 28 — Triggers

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## Consequences

---

In order to maintain *data integrity*, sometimes an operation on one table has consequences for the values in another table.

One example of this is *referential integrity* where declaring foreign keys enables changes in a parent table to be cascaded.

However there are other aspects to data integrity, and *triggers* provide a very general mechanism for ensuring that the data in multiple tables remains consistent.

# Triggers

---

*Triggers* are SQL statements that are *run automatically* when the data in a nominated table changes.

Each trigger is defined on a *named table*, and so we can think of a trigger as “watching out” for changes in that table.

# Trigger Syntax

---

```
CREATE TRIGGER triggerName timeSpec triggerAction  
ON tableName  
FOR EACH ROW  
BEGIN  
...  
END
```

In this template,

- ▶ `triggerName` is the name of the *trigger*,
- ▶ `timeSpec` and `triggerAction` must be replaced by appropriate keywords,
- ▶ `tableName` is the name of the *table* being monitored.

## Actions

---

The triggerAction indicates *what action* will cause this trigger to *fire* (i.e., be executed).

It must be one of:

- ▶ DELETE
- ▶ INSERT
- ▶ UPDATE

A **DELETE** trigger will be fired whenever a **DELETE** statement is executed on the table, an **INSERT** trigger will be fired whenever an **INSERT** operation is executed, and similarly for **UPDATE**.

## Times

---

The `timeSpec` specifies exactly *when* the trigger will fire

It must be one of:

- ▶ `BEFORE`
- ▶ `AFTER`
- ▶ `INSTEAD OF` (only on views)

A `BEFORE` trigger will run *before* the specified operation occurs, an `AFTER` trigger will run *after* the specified operation occurs.

An `INSTEAD OF` trigger will run *instead of* the originally specified operation (but in SQLite this only works on views).

## A log file

---

Suppose we have a table representing bank accounts

```
CREATE TABLE BankAccount (
    accountNum INTEGER PRIMARY KEY,
    accountName TEXT,
    accountBalance REAL);
```

Make a table to *maintain a log* of events on BankAccount.

```
CREATE TABLE EventLog (
    eventTime TEXT);
```

## Add a trigger

---

We can't rely on users to log their own actions, so it needs to be something that happens automatically:

```
CREATE TRIGGER EventLogger BEFORE UPDATE  
ON BankAccount  
BEGIN  
INSERT INTO EventLog VALUES (datetime());  
END;
```

This trigger is fired immediately *before* an *update* is made on *BankAccount*.

# In SQLiteStudio

---

```
74 CREATE TRIGGER Event BEFORE UPDATE
75 ON BankAccount
76 BEGIN
77 INSERT INTO EventLog VALUES (datetime());
78 END;
79
80 INSERT INTO BankAccount VALUES(1, 'Gordon', 1000);
81
82 UPDATE BankAccount SET accountBalance = 1010;
83 UPDATE BankAccount SET accountBalance = accountBalance*1.05;
84 UPDATE BankAccount SET accountBalance = accountBalance*1.05;
85
86 SELECT * FROM EventLog;
```

Grid view

Total rows loaded: 3

	eventTime
1	2020-09-15 08:11:30
2	2020-09-15 08:12:01
3	2020-09-15 08:12:06

## Accessing the row being changed

---

To do something more realistic, such as logging the actual account number, the trigger needs to *know the values* in the row that is changing.

These are stored in special tuples called **NEW** and **OLD**.

If the statement is an **UPDATE** statement then

- ▶ **OLD** contains the values of the row before the update
- ▶ **NEW** contains the values of the row after the update

## Better logging

---

This event log will track events, but now with the additional information about the balance of the account before and after the change.

```
CREATE TABLE BetterEventLog (
    accountNum INTEGER,
    preBalance REAL,
    postBalance REAL,
    eventTime TEXT);
```

## A better trigger

---

```
CREATE TRIGGER BetterEventLogger BEFORE UPDATE
ON BankAccount
BEGIN
INSERT INTO BetterEventLog VALUES
( NEW.accountNum, OLD.accountBalance,
    NEW.accountBalance, datetime() );
END;
```

Seeing the `accountNum` does not change, we could use either `NEW.accountNum` or `OLD.accountNum`.

## Deposits or withdrawals?

---

Suppose we also wish to add another field to the event logger to indicate whether this is a deposit or a withdrawal.

```
CREATE TABLE BestEventLog (
    accountNum INTEGER,
    preBalance REAL,
    postBalance REAL,
    eventType TEXT,
    eventTime TEXT);
```

We want to set eventType to either Deposit or Withdrawal

## How do we test?

---

This seems to require some “if-then-else” programming logic

- ▶ It is a *deposit* if

```
OLD.accountBalance < NEW.accountBalance
```

- ▶ It is a *withdrawal* if

```
OLD.accountBalance > NEW.accountBalance
```

Although SQL is not a procedural programming language, different variants of SQL have some procedural programming constructs.

## The CASE statement

---

The **CASE** statement (similar to C switch) is an expression whose value depends on boolean expressions included in the expression.

```
CASE
    WHEN <bool1> THEN <expr1>
    WHEN <bool2> THEN <expr2>
    ...
    ELSE <expr>
END
```

The value of this expression is the first **<expr>** for which the boolean condition is true.

## For event logging

---

This is an expression that takes on the values 'Deposit', 'Withdrawal' or NULL depending on the value of boolean expressions.

```
CASE
    WHEN OLD.balance < NEW.balance THEN 'Deposit'
    WHEN NEW.balance < OLD.balance THEN 'Withdrawal'
    ELSE NULL
END
```

## The final trigger

---

```
CREATE TRIGGER BestEventLogger BEFORE UPDATE
ON BankAccount
BEGIN
INSERT INTO BestEventLog VALUES (
    NEW.accountNum,
    OLD.accountBalance,
    NEW.accountBalance,
    CASE
        WHEN NEW.accountBalance > OLD.accountBalance THEN 'Deposit'
        WHEN NEW.accountBalance < OLD.accountBalance THEN 'Withdrawal'
        ELSE NULL
    END,
    datetETIME());
END ;
```

## Trigger uses

---

This may seem to be a trivial example, but triggers can

- ▶ Validate and abort actions if they violate some business rule
- ▶ Maintain auxiliary databases to simulate *materialized views*
- ▶ Implement access control (not so much in SQLite)

The power to intercept an arbitrary operation on a database, and (programmatically) make decisions about how to proceed adds an additional flexible layer of security and integrity.

# CITS1402

## Relational Database Management Systems

### Video 30 — Indexes

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## A common situation

---

Consider a *very large table* relating *names* to *numbers* which is *alphabetically* ordered.<sup>1</sup>

name	num
'Aardvark'	45813012
'Aarons'	21342214
'Aakon'	12308082
.....	.....
'Zzypo'	12313243

In the experiments below I used a table like this with  $2^{24}$  rows (this is about 16 million rows).

---

<sup>1</sup>A “telephone directory” is an old-fashioned paper-based implementation of such a database.

## Which is easier

---

Because the computer knows how the table has been stored in memory, it can locate the values in a particular row (given by row number) without any searching.

Given this, which of the following is easier for a computer?

- ▶ Find the *number* associated to the person 'Snodgrass'
- ▶ Find the *name* associated to the number 15484722

## Counting probes

---

A *probe* is the action of *finding* (on the disk), *reading* one of the rows of the table and testing it to see if contains the right name or right number.

The time taken to complete the query is determined by the *number of probes* required.

Can we use the knowledge that *name* is alphabetically ordered to find information using fewer probes?

## From number to name

---

Who has number 12540393?

```
SELECT *
FROM indexTest
WHERE num = 12540393;
```

The system has to perform a *linear search*, and probe *each row in turn* until it finds the right one(s).

If numbers are not unique, then  $2^{24}$  probes are required for every search.

## Names to numbers

---

What is the number for 'Snodgrass'.

The computer knows that any row containing 'Snodgrass' must lie somewhere between row 1 and row  $2^{24}$ .

*Probe the row* that is *half-way* between these two extremes, that is, look at row  $2^{23}$ .

If this row has `name` equal to, say, 'Martin' then we know that we only need to search between rows  $2^{23}$  and  $2^{24}$ .

If this row has `name` equal to, say, 'Thomas' then we know that we only need to search between rows 1 and  $2^{23}$

## Binary Search

---

This search proceeds by identifying a range of rows that must contain 'Snodgrass' and then *repeatedly halving* that range by probing the middle row.



With  $2^{24}$  values we only need 24 probes instead of  $2^{24}$ .

## General performance

---

If the database has  $n$  rows then

- ▶ `num` to `name` takes about  $n/2$  probes
- ▶ `name` to `num` takes about  $\log_2(n)$  probes

It is hard to overstate the performance gained by replacing an operation that is linear in  $n$  with one that is logarithmic in  $n$ .

# An index

---

An *index* on a *particular column* in a database table is an *auxiliary data structure* that speeds up searches on *that column* from linear to logarithmic.

```
CREATE TABLE indexTest (name TEXT, num INTEGER);
```

This table has no indexes at present.

# Some experiments

---

```
sqlite> .timer on
sqlite> SELECT COUNT(*) FROM indexTest;
16777225
Run Time: real 1.832 user 0.140657 sys 0.406743
sqlite> SELECT * FROM indexTest LIMIT 10;
fnaouvzn|40025205
xuosakzx|89647276
kpstjxdh|78781003
jjasgseal|39406548
kefilnzf|83604876
oxwbkyco|59485848
dszkzrpr|2863024
izyyzaoj|54355471
tmpsfqlp|32830374
vtcfkixd|60496005
Run Time: real 0.000 user 0.000148 sys 0.000311
```

## Finding values

---

```
sqlite> SELECT * FROM indexTest WHERE name = 'jpwqtldq'  
jpwqtldq|43727921  
Run Time: real 1.106 user 0.982750 sys 0.118336  
sqlite> SELECT * FROM indexTest WHERE num = 43727921;  
jpwqtldq|43727921  
Run Time: real 0.967 user 0.843605 sys 0.115537
```

## Adding an index

---

If we know that looking up a value in a particular column is frequently-needed, then we can *add an index* to that column.

```
CREATE INDEX <index_name>
ON <table_name>(<column_name>);
```

This constructs a special data-structure, known as a *balanced tree*, or B-tree, and stores it in the database file.

The location of a value in the B-tree can be found in about  $\log_2(n)$  operations.

(It is not exactly binary search, but the overall effect is similar)

# How to think about it

---

aaaabzgh	12018128
aaaacccm	1155289
aaaacmoi	5237408
aaaacwdx	5064740
aaaacyla	12265617
aaaadiwu	252019
aaaadxzs	3754987
aaaageged	6391242
aaaaeuxc	9212731
aaaafhex	11865182
aaaafqkq	331957
aaaaggry	12046824
aaaaitbf	8649402
aaaajeui	9550183
aaaakdrc	5367638
aaaakrga	14900265
aaaaldog	4673247
aaaamjsl	12409351
aaaamlsj	11562891
aaaamrgp	7340566

You can view creating an index as *similar* (not identical) to “creating a phone-book” for that particular column.

Whenever SQL needs to search for rows with a particular value of `name`, then it can very quickly “look up” the row and hence the value.

# Speed vs Disk

---

Computing the index takes quite a bit of *time*:

```
sqlite> CREATE INDEX nameIndex ON indexTest(name);
Run Time: real 40.479 user 19.170171 sys 4.265781
sqlite>
```

Storing the newly computed index takes quite a bit of (disk) *space*.

```
00013890@DEP52010 Index % du -sh indexTest.db
351M    indexTest.db
00013890@DEP52010 Index % du -sh indexTest.db
639M    indexTest.db
```

# Using the index

---

Queries on the *indexed column* are now *blindingly fast* — dropped from one second to a few 1/1000ths of a second.

```
sqlite> SELECT * FROM indexTest WHERE name = 'lmnrogjo';
lmnrogjol20751656
Run Time: real 0.003 user 0.000114 sys 0.000245
sqlite> SELECT * FROM indexTest WHERE num = 20751656;
lmnrogjol20751656
Run Time: real 0.935 user 0.808253 sys 0.118248
```

Queries on the *non-indexed column* are still *glutinously slow*.

## Updating an index

---

An index must be *updated* whenever the data changes.

This is done automatically, but it means that every `INSERT` and `DELETE` will be slower than if the table had no indexes.

So if the data is

- ▶ Large or very large
- ▶ Frequently queried
- ▶ Infrequently changing

then *add an index* to one or more columns.

## Finding an index

---

To find out if a table has an index defined on it.

```
sqlite> .schema indexTest
CREATE TABLE indexTest(name TEXT, num INTEGER);
CREATE INDEX nameIndex ON indexTest(name);
```

An index is viewed as part of the *schema* (i.e., structure) of the table.

## Cleaning up

---

Indexes are deleted with the usual `DROP` command, using the name that the index was given.

```
DROP INDEX <index_name>;
```

This drops the index but does not recover the space it used on the disk, so you have to do some “housework” to get the space back.

```
sqlite> DROP INDEX nameIndex;  
Run Time: real 1.136 user 0.173355 sys 0.229342  
sqlite> VACUUM;  
Run Time: real 6.928 user 2.141986 sys 4.247055
```

## Automatic Indexes

---

An index is automatically created on a **PRIMARY KEY** column.

The primary key uniquely identifies a row in the table, so it is extremely likely to be:

- ▶ involved in **JOIN** statements,
- ▶ involved in **FOREIGN KEY** constraints,
- ▶ used in subqueries.

## Composite Indexes

---

In addition to indexing a *single* column, it is possible to index *combinations* of columns.

```
CREATE INDEX <index_name>
ON <table_name>(C1, C2);
```

Queries that involve *both* of those columns (at the same time) will be vastly sped up.

Queries that involve *only C1* will be sped up, but queries that involve only *C2* will not.

# Composite Index

---

Suppose that a table has a composite index

```
CREATE INDEX doubleIdx ON DataTable (A, B);
```

Then the data in the index is organised first according to **A** and then according to **B *within*** each value of **A**.

- ▶  $A = 10$ 
  - ▶  $B = 1, B = 2, B = 5, B = 10$
- ▶  $A = 18$ 
  - ▶  $B = 5, B = 7, B = 8, B = 11$
- ▶  $A = 40$ 
  - ▶  $B = 15, B = 17, B = 20, B = 25$

# Queries

---

Consider three queries:

```
SELECT * FROM <table>
WHERE A =18 AND B = 7;
```

```
SELECT * FROM <table>
WHERE A = 10;
```

```
SELECT * FROM <table>
WHERE B = 7;
```

The composite index will help on the first two, but not the third.

# CITS1402

## Relational Database Management Systems

### Video 31 — Relationships

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Relationships

---

This video goes into more detail about *relationships*,

- ▶ Types of relationship
- ▶ Relationships in an ERD
- ▶ Relationships in SQL

## What is a relationship?

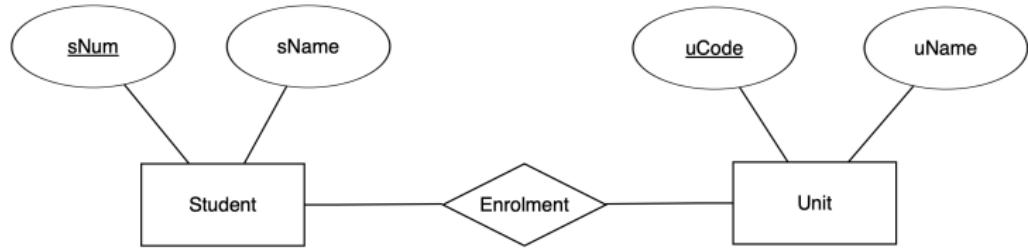
---

Given two sets  $X$ ,  $Y$ , a (binary) relationship is a *set of pairs*  $(x, y)$  where  $x \in X$ , and  $y \in Y$ .

For example, we can take

- ▶  $X$  to be the set of students
- ▶  $Y$  to be the set of units
- ▶  $(x, y)$  is in the relation if  $x$  is enrolled in  $y$ .

This is the relation we have been calling Enrolment.



## The relation

---

The relation can be given just as a list of pairs

(1, 'CITS1402')

(1, 'CITS1401')

(1, 'MATH1001')

(2, 'CITS1402')

(2, 'CITS1401')

We've already seen that in an RDBMS, we *can store* a relation as a table with two columns that are foreign keys, along with any *relationship attributes*.

## Types of relationship

---

In the Enrolment relation:

- ▶ One Student can enrol in *multiple* Units
- ▶ One Unit can have *multiple* Students enrolled

(1, 'CITS1402')

(1, 'CITS1401')

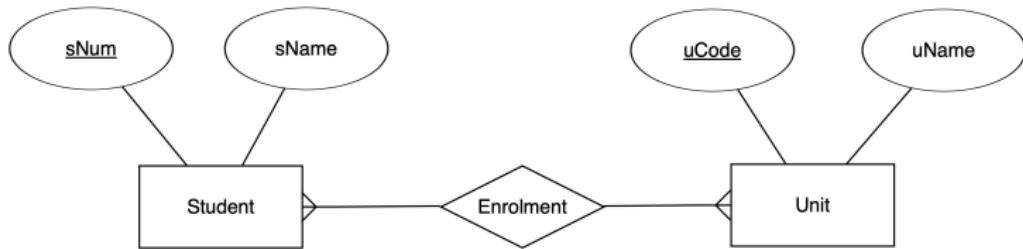
(1, 'MATH1001')

(2, 'CITS1402')

(2, 'CITS1401')

## In the ERD

---



A *many-to-many* relation in the ERD is implemented as a table in the database.

# One-to-many relationships

---

There are many situations that involve *one-to-many* relationships.

Hugo

Frank

Ethel

Dan

Chan

Jan

Bill

Paris

London

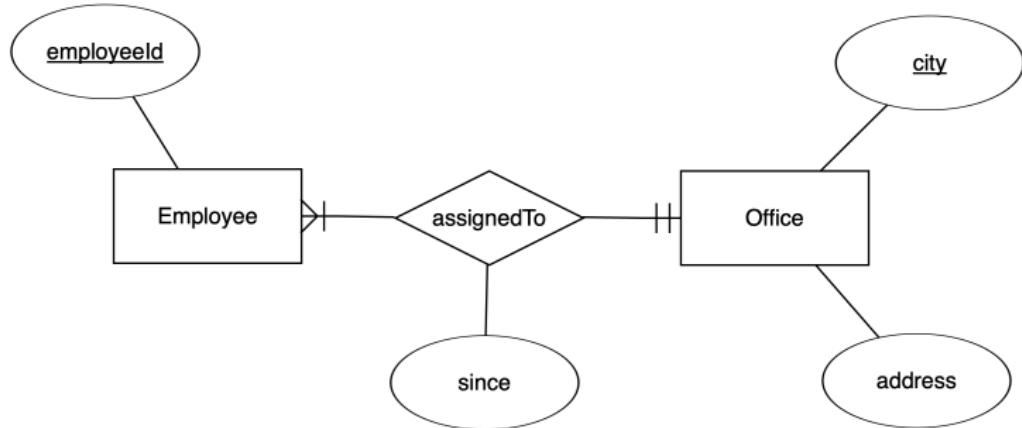
Sydney

EMPLOYEES

OFFICES

# ClassicModels

---



## Implementing this relationship

---

In the *Classic Models* database, there is no table corresponding to `assignedTo`, but instead the table `Employees` has a column `officeCode` which is a *foreign key* to the table `Office`.

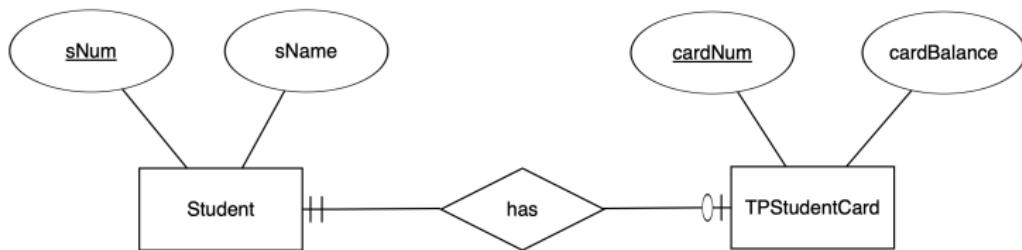
This is only possible because the *business logic* of the company states that each employee is assigned to *exactly one* office.

If the relationship is *optional*, then it still possible to implement it as a foreign key, because in SQLite, a foreign key column can be `NULL`.

# One-to-one relationship

---

The final type of relationship is *one-to-one*.



Each student has *at most one* TransPerth Student Card, and each TransPerth Student Card belongs to *exactly one* student.

## One-to-one relationship

---

A one-to-one relationship with *mandatory participation* by one entity would normally be implemented using a foreign key on the corresponding table.

```
CREATE TABLE TPStudentCard (
    cardNum INTEGER PRIMARY KEY,
    cardBalance REAL,
    sNum INTEGER,
    FOREIGN KEY (sNum) REFERENCES Student(sNum));
```

## Summary

---

After the ERD is completed, it is translated into a relational schema, where usually

- ▶ Entity sets become tables,
- ▶ Many-to-many relationships become tables,
- ▶ 1-to-many relationships usually become foreign keys,
- ▶ 1-to-1 relationships become foreign keys.

CITS1402  
Relational Database Management Systems

Video 32 — Functional Dependencies II

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Functional Dependencies

---

In Video 21, we discussed “splitting” a relation (table) into two separate relations, such as the following table for student marks.

```
CREATE TABLE R (S INT,  
N TEXT,  
U TEXT,  
M INT);
```

Each student will appear in multiple rows, once for each unit.

The *association* between a student's number and their name is therefore *repeated* many times.

## Decomposition

---

In this case it seems “obvious” that instead of

$R(S, N, U, M)$

we should have

$R1(S, N)$

$R2(S, U, M)$

So  $R1$  keeps track of “number-to-name” information, while  $R2$  keeps track of “marks-per-unit” information.

## In general

---

The *desired* relationship between  $R$ ,  $R_1$  and  $R_2$  is clear:

- ▶ *Projection* is used to go from  $R \rightarrow R_1, R_2$
- ▶ *Joining* is used to go from  $R_1, R_2 \rightarrow R$

It is loosely analogous to storing a number by its *factors* rather than directly

$$35 = 5 \times 7$$

$$R = R_1 \bowtie R_2$$

## Lossless-join decomposition

---

If a relation  $R$  is decomposed into relations  $R_1$ ,  $R_2$  such that *for every legal instance  $r$  of  $R$*

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r)$$

then the *decomposition itself* is said to be a *lossless-join* decomposition.

You can view this as a sort of *minimum requirement* for a decomposition to be acceptable.

## Not lossless-join

---

Take the same  $R = SNUM$ , but this time decompose as

$S1 = SM$

$S2 = NUM$

If we can find *even one example* where we cannot recover  $S$  from  $S1$  and  $S2$  then this is *not* lossless-join.

On the other hand, to show that a decomposition *is* lossless-join needs a *logical argument*.

## Not lossless-join

---

```
DROP TABLE IF EXISTS S;
CREATE TABLE S (S INT, N TEXT, U TEXT, M INT );
INSERT INTO S VALUES
    (1, 'Amy', 'CITS1401', 72),
    (1, 'Amy', 'CITS1402', 68),
    (1, 'Amy', 'MATH1001', 68),
    |(2, 'Bill', 'CITS1401', 88),
    (2, 'Bill', 'CITS1402', 68);

DROP TABLE IF EXISTS S1;
DROP TABLE IF EXISTS S2;
CREATE TABLE S1 (S INT, M INT );
CREATE TABLE S2 (N TEXT, U TEXT, M INT );

INSERT INTO S1 SELECT DISTINCT S, M FROM S;
INSERT INTO S2 SELECT DISTINCT N, U, M FROM S;
```

## The two “factors”

---

S	M
1	72
1	68
2	88
2	68

N	U	M
Amy	CITS1401	72
Amy	CITS1402	68
Amy	MATH1001	68
Bill	CITS1401	88
Bill	CITS1402	68

# The “product”

---

```
3 SELECT S, N, U, M  
4 FROM S1 NATURAL JOIN S2;
```

Grid view

S	N	U	M
1	Amy	CITS1401	72
1	Amy	CITS1402	68
1	Amy	MATH1001	68
1	Bill	CITS1402	68
2	Bill	CITS1401	88
2	Amy	CITS1402	68
2	Amy	MATH1001	68
2	Bill	CITS1402	68

There are “surplus rows” that was not in the original table.

## Why did this happen?

---

The **NATURAL JOIN** joins the tables on the M column, because it is the only column that  $S_1$  and  $S_2$  have in common.

S	M
1	72
1	68
2	88
2	68

N	U	M
Amy	CITS1401	72
Amy	CITS1402	68
Amy	MATH1001	68
Bill	CITS1401	88
Bill	CITS1402	68

But in  $S_1$  there are *distinct rows* with the same value of M and in  $S_2$  there are *distinct rows* with the same value of M

# Keys

---

In a database, a *key* is an *attribute* or a *set of attributes* with the property that only one row in the database can have the same values for those attributes.

In the  $R(S, N, U, M)$  example, we have

- ▶ SNU is a key
- ▶ SU is a key
- ▶ S is not a key

A key is a *minimal key* if no *proper subset* of its attributes is a key.

## Key terminology

---

There is a huge mess around the terminology for keys.

- ▶ Key
- ▶ Superkey
- ▶ Candidate key
- ▶ Minimal key

Note that identifying keys involves the “business logic” of the table, and is not determined by the *current contents* of the table.

## Lossless-join decomposition

---

A decomposition of  $R$  into  $R_1$  and  $R_2$  is *lossless-join* if and only if

- ▶  $R_1 \cup R_2 = R$ , and
- ▶  $R_1 \cap R_2$  is a *key* for either  $R_1$  or  $R_2$ .

In the example,

$$R = \text{SNUM}$$

$$R_1 = \text{SN}$$

$$R_2 = \text{SUM}$$

$$R_1 \cap R_2 = S$$

and  $S$  is a key for  $R_1$ .

# Normalization

---

There is a tension between *redundancy* and *efficiency* in RDBMS.

- ▶ *Normalization* is the process of *decomposing* individual tables into two or more smaller (i.e, fewer columns) tables.
- ▶ *Denormalization* is the process of *reconstructing* data by joining multiple smaller tables into fewer larger ones.

A large number of small tables is good for reducing redundancy, but constantly making huge multi-table joins is difficult to code, resource-intensive to run, and doesn't scale out well.

The theory of normalization defines different *types* and *levels* of redundancy, allowing users to make an *informed choice*.

# CITS1402

## Relational Database Management Systems

### Video 33 — Boyce-Codd Normal Form

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



THE UNIVERSITY OF  
**WESTERN**  
AUSTRALIA

# Keys

---

Recall that a *key* for a table is a column (or a combination of columns) that *uniquely determines* a row.

Business logic *identifies* the keys, and the relation schema *enforces* the business logic.

```
CREATE TABLE Student (
    num INTEGER PRIMARY KEY,
    name TEXT,
    address TEXT);
```

## Composite keys

---

Sometimes it requires a *combination* of columns to uniquely determine a row.

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

## Some odd terminology

---

The fundamental property of a key is that *just the key value* is enough to locate the *only row* with that particular value.

In the university database, I can look up a student's *name* from their student number alone.

This situation is described by the odd phrase:

*"student number determines name"*

This really means "there cannot be two rows with the same student number, but different names".

## Functional dependencies

---

A *functional dependency* (an FD) is a generalization of the concept of a *key* in a relation.

Suppose that  $X$  and  $Y$  are two *subsets* of the attributes of a relation with the following property:

*“No two tuples can be identical on  $X$ , but different on  $Y$ ”*

In this situation we say that  $X$  *determines*  $Y$  and write

$$X \rightarrow Y.$$

In particular, we *can have* two rows identical on  $X$ , but then they have to be identical on  $Y$ .

# Functional Dependencies

---

Id (I)	Name (N)	Level (L)	Rate (R)
1	Smith	10	55.00
2	Jones	8	30.00
3	Tan	10	55.00
4	White	9	42.00
:	:	:	

One easy functional dependency is

$$L \rightarrow R$$

because any two people at the same level get the same rate, i.e.,  
*“the level determines the rate”.*

## Keys

---

Another FD is given by

$$I \rightarrow INLR$$

meaning that *every attribute* in the table is determined by  $I$ .

This is true because  $I$  is actually a *key* for the table and so the value of  $I$  identifies (at most) one row.

The functional dependencies in a table give hints as to suitable lossless-join decompositions.

## Normal forms

---

Boyce and Codd considered various restrictions that could be imposed on the functional dependencies of a database schema.

They defined a hierarchy of *increasingly restrictive* conditions on FDs, and introduced specific terminology to describe schemas that meet each level of conditions.

These levels are called *normal forms*.

# Normal form hierarchy

---

- ▶ **1NF (First normal form)**

Entries in the table are *scalar values* (not sets)

- ▶ **2NF (Second normal form)**

1NF *plus* every non-key attribute depends on the whole key  
(only relevant where there are composite keys)

- ▶ **3NF (Third normal form)**

2NF *plus* no transitive dependencies

- ▶ **BCNF (Boyce-Codd normal form)**

3NF *plus* conditions to be discussed

## BCNF

---

A relational schema is in *Boyce-Codd normal form* if for every functional dependency  $X \rightarrow A$  (where  $X$  is a subset of the attributes and  $A$  is a single attribute) either

- ▶  $A \in X$  (that is,  $X \rightarrow A$  is a trivial FD), or
- ▶  $X$  is a key.

In other words, the **only functional dependencies** are either the trivial or ones based on the keys of the relation.

If a relational schema is in BCNF, then there is **no redundancy** arising from functional dependencies.

## No redundancy in BCNF

---

A relational schema in BCNF is already in its “leanest” form:

Each attribute is determined only by the whole key, so nothing can be deduced from fewer values.

The relation **INLR** given above is not in BCNF because  $L \rightarrow R$  is a functional dependency.

## BCNF decomposition

---

Suppose a relation  $R$  is *not* in BCNF. Then there must be some functional dependency

$$\textcolor{red}{X} \rightarrow \textcolor{red}{Y}$$

where  $\textcolor{red}{X}$  is not a key; this is a *Boyce-Codd violation*.

We can assume that  $\textcolor{red}{Y} \cap \textcolor{red}{X} = \emptyset$  so  $\textcolor{red}{Y}$  only contains some “extra” attributes determined by  $\textcolor{red}{X}$ , not the ones in  $\textcolor{red}{X}$  itself.

Then the relation can be *decomposed* into the two relations

$$R_1 = R - \textcolor{red}{Y} \quad R_2 = \textcolor{red}{XY}$$

As  $\textcolor{red}{X}$  is a key for  $R_2$ , this is a lossless-join decomposition.

## The decomposition

---

In the **INLR** example, we have

$$L \rightarrow R$$

and so the rule says

$$R_1 = \text{INL} \quad R_2 = \text{LR}$$

which is exactly the decomposition we found earlier.

## Reasoning about FDs

---

Some FDs are obvious from the *semantics*<sup>1</sup> of a relation, while others may follow as a *consequence* of these initial ones.

For example, if  $A \rightarrow B$  and  $B \rightarrow C$ , then it follows that

$$A \rightarrow C.$$

(Take two tuples with the same values for attribute  $A$ , then they must have the same values for attribute  $B$  because of the first FD, and so they must have the same values for  $C$  by the second FD.)

---

<sup>1</sup>i.e. the *meaning* of the attributes

## Example

---

Suppose  $R(A, B, C, D, E)$  has the following FDs

$$D \rightarrow C, \quad CE \rightarrow A$$

Is  $BDE$  a key for  $R$ ?

- ▶ So far we know that  $BDE \rightarrow BDE$
- ▶ From the FD  $D \rightarrow C$  it follows that  $BDE \rightarrow BCDE$
- ▶ From the FD  $CE \rightarrow A$  it follows that  $BDE \rightarrow ABCDE$

Therefore two rows that agree on  $BDE$  must agree on all the columns so  $BDE$  is a key.

## BCNF decomposition cont.

---

If either  $R_1$  or  $R_2$  is not in BCNF then the process can be continued, by decomposing them in the same fashion.

By continually decomposing any relation not in BCNF into smaller relations, we must eventually end up with a collection of relations that are in BCNF.

Therefore any initial schema can be decomposed into BCNF.

# Is BCNF the ultimate answer?

---

Definitely not!

There are various problems with BCNF:

- ▶ While it reduces redundancy, queries may take considerably longer, as they now involve possibly complicated joins
- ▶ Some FDs that hold on the original relation can no longer be enforced using the decomposed relations

There are numerous other *normal forms* each of which has an associated decomposition theory, and choosing whether and how to decompose is an important task for the database designer.

# CITS1402

## Relational Database Management Systems

### Video 34 — Transactions

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



THE UNIVERSITY OF  
**WESTERN**  
**AUSTRALIA**

# ACID

---

**ACID** is one of the most famous acronyms universally associated with (relational) database management systems.

The four words refer to four *desirable properties* of a database, all related to ensuring the database is always in a *valid state*.

- ▶ Atomicity
- ▶ Consistency
- ▶ Isolation
- ▶ Durability

A database is called ACID-compliant if it has these properties.

## Videos

---

You may find Jennifer Widom's videos on transactions useful:

<https://youtu.be/FA85kHsJss4>

<https://youtu.be/aNCpEC0-VVs>

There is a third video but not in the scope of the unit this year.

<https://youtu.be/kTxxh-XPUK4>

## Robustness

---

One of the most important properties of a modern DBMS is that it is *robust* under both normal and unusual operating conditions.

- ▶ Normal conditions include multiple users *concurrently accessing* the database.
- ▶ Unusual conditions include computer crashes, connection failures, disk full errors and power outages etc.

It is important for a DBA to understand the role that *transactions* play in this robustness.

## Multiple clients

---

Most databases are being used by more than one client simultaneously.

For example, there may be tens or even hundreds of people using a ticket website to buy tickets for upcoming events.

Normally the statements will just be run in whatever order they arrive at the database.

If all the statements are just *queries* then this does not matter.

*But what if the statements alter the database?*

## Multiple requests

---

Suppose that user  $A$  makes statements  $S_1, S_2, S_3$ , and user  $B$  makes statements  $T_1, T_2, T_3$ .

In order to keep the system responsive for all users, the system will *interleave* the statements from  $A$  with the statements from  $B$ . So the system might actually run:

$S_1, T_1, T_2, S_2, S_3, T_3$

in that order.

What if  $A$  and  $B$  are trying to work with the *same table* and some of  $A$ 's statements are altering things that  $B$  needs?

## Bank transfers

---

The “canonical example” of an application where correct treatment of transactions is critical is transferring money in a bank.

For example, suppose that a user at an ATM *transfers money* from account 1 to account 2 — this will require *two* SQL statements.

```
UPDATE Accounts
SET balance = balance - 500
WHERE id = 1;
```

```
UPDATE Accounts
SET balance = balance + 500
WHERE id = 2;
```

# System failure

---

If the system crashes *between* the statements, then the database is in an *inconsistent state*.

We need to tell SQL that these two statements *belong together*, and that the system must perform *both* or *neither* of these statements.

```
UPDATE Accounts  
SET balance = balance - 500  
WHERE id = 1;
```

```
UPDATE Accounts  
SET balance = balance + 500  
WHERE id = 2;
```

# Transactions

---

A *transaction* is defined to be a sequence of statements that comprise a meaningful activity in the user's environment.

For example, the bank transfer consists of two statements that “belong together” to complete a meaningful task.

Transactions codified in the 1983 paper *“Principles of Transaction Oriented Database Recovery”*, by Härdler & Reuter.

They observed that thinking of a user's interaction with the database as a *sequence of transactions* enabled the development of robust protocols for maintaining the integrity of the database.

## Atomicity

---

The word *atomic* means *indivisible*.

In a DB context, transactions are *atomic* if the system ensures that they cannot be “half-done” — in other words, the system guarantees that either *the entire transaction* completes or it fails and has *no effect* on the database.

The bank transfer example above is one application where users rely on the atomicity of transactions.

## Consistency

---

Transactions must preserve the *consistency* of the database.

More precisely, if the database is in a consistent state, and a transaction is executed, then the database should end up in a consistent state.

Some database changes have *consequences* (e.g., foreign key cascades) and these consequences must be fully executed before the statement is deemed complete.

## Isolation

---

*Isolation* means that each user of the DB should be able to execute a transaction *as though* they are the only user.

In other words, each user's actions should be *isolated* from the actions of other users.

Multi-user systems cannot afford to *actually* isolate transactions nor can they *interleave* statements arbitrarily.

A database implements *locking protocols* (on a table or even a row) to prevent interference.

## Durability

---

*Durability* means that once the user is informed of the successful completion of a transaction, then its effects on the database are persistent.

Thus the user should be shielded from any possible problems (eg system crashes) that might occur after being notified that the transaction has successfully completed.

## Atomicity and Durability

---

To ensure *atomicity* the DBMS must be able to *roll back* the database if the transaction fails for any reason.

- ▶ The DBMS maintains a *log* of all changes
- ▶ Each statement is *first* recorded in the log file
- ▶ The database is then updated and written to disk

The property that changes are logged before they are actually made on disk is called a *write ahead log*.

## Implicit transactions

---

Most transactions are *implicit*, in that in each statement is considered to occur in its own transaction.

```
UPDATE Enrolled  
SET mark = mark + 5  
WHERE sNum = 2  
    AND uCode = 'CITS1402'  
    AND yr = 2022;
```

This requires the computer to *read* the value, *perform the addition* and then *write* the new value.

What if the system crashes half-way through?

# SQLite

---

In SQLite a transaction can be initiated by a user:

`BEGIN TRANSACTION;`

All statements after that will be deemed to form part of the same transaction until one of the statements

- ▶ `COMMIT;`
- ▶ `END TRANSACTION;`
- ▶ `ROLLBACK;`

CITS1402  
Relational Database Management Systems

Video 35 — MongoDB

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# NoSQL

---

The term NoSQL stands for

- ▶ Not SQL, or
- ▶ Not *only* SQL.

It is used to describe a range of database management systems that do not use the *relational model of data*.

The most prominent NoSQL database is MongoDB which is a *document* database.

## The hype

---

NoSQL is often hyped as

*“a modern, flexible and scalable solution for today’s massive and rapidly-changing cloud-based datasets”.*

On the contrary, SQL is described as being

*“A legacy system that is difficult to use, inflexible and does not scale out well.”*

*NoSQL is portrayed as the “cool creative hipster” disrupting the fossilized stuffy world of SQL.*

# Structure

---

The main difference is whether to view data as *highly structured* or *loosely structured*.

- ▶ In SQL, data is always *structured* into tables, and new data must fit into the existing structure
- ▶ In MongoDB, data is largely *unstructured*, and new data of any type can be added anywhere at any time

# Data in RDBMS

---

In an RDBMS, tables create a *strict template* for data.

- ▶ Each table has a fixed *schema*, which determines the number, type and name of all the columns
- ▶ The schema is designed in advance, and changing it is difficult
- ▶ Database is normalized to minimise redundancy and storage requirements

## Extracting data

---

The data about a single entity (i.e., one particular **Student**) may be split over a large number of tables.

Extracting this data may require a cumbersome multi-table join that is difficult to write and understand, and possibly time-consuming to execute.

The cost of disk storage relative to processing power has reduced **dramatically** over the last few decades.

# JSON

---

JavaScript Object Notation (JSON) is a text-based *key-value* format used to store and exchange data objects.

```
{  
    id: 1,  
    name: "Amy",  
    addresses: [  
        { add: "3 Wallaby Rd", postCode: 3002},  
        { add: "66 Broadway", postCode: 6009}  
    ],  
    unitmarks: [  
        { unit: "CITS1402", name: "Databases", mark: 88},  
        { unit: "MATH1011", name: "Maths 1", mark: 66}  
    ]  
}
```

A single object (as above) is called a *document*.

# JSON

---

Each document consists of *fields* and each field consists of a *key*<sup>1</sup> and a *value*.

For example, `id` is a key and `1` is its value.

The *value* can be a single value or an *array*.

The elements of the array can themselves be *documents*.

---

<sup>1</sup>do not get confused with keys in a relational database

## Key-value

---

In this document we have

Key	Value
id	1
name	"Amy"
addresses	<i>an array</i>
unitmarks	<i>an array</i>

Each array contains further *embedded documents*.

## Documents

---

A *document* can contain any number of key/value pairs.

Values can be primitives, documents or arrays (of anything).

In general, a document can be *arbitrarily complicated*.

In a document database, *all of the information* about an entity can be stored in a *single document*.

# Fractal documents

---



By Jon Sullivan <https://commons.wikimedia.org/w/index.php?curid=95997>

# The database

---

In MongoDB,

- ▶ A database contains *collections*
- ▶ A collection contains *documents*
- ▶ Documents contain *key-value* pairs

Sophisticated *indexing* and *search algorithms* make retrieving data very fast.

# Correspondence

---

The correspondence between document and relational databases is:

Document DB	Relational DB
Database	Database
Collection	Table
Document	Row

## Movies

---

The Internet Movie Database (IMDB) stores a vast amount of movie-related information; the data format is described at <https://www.imdb.com/interfaces/>.

It stores the data in files that are basically relational in nature.

- ▶ The file `title.basics.tsv` stores information about *movies*
- ▶ The file `name.basics.tsv` stores information about *people*
- ▶ The file `title.principals.tsv` is a *relationship* between people and movies

# Who starred in Jurassic Park?

---

In the IMDB database, the title.basics table contains:

```
tt0107290 movie Jurassic Park 1993 Action,Adventure,Sci-Fi
```

The title.principals table contains:

```
tt0107290 nm0002354 composer
tt0107290 nm0000554 actor
tt0107290 nm0000368 actress
tt0107290 nm0000156 actor
tt0107290 nm0000277 actor
tt0107290 nm0000229 director
```

The names.basics table contains

```
nm0000554 Sam Neill 1947
nm0000368 Laura Dern 1967
nm0000156 Jeff Goldblum 1952
nm0000277 Richard Attenborough 1923
```

# In MongoDB

---

In MongoDB, everything is in a single document:

```
TER {"title":"Jurassic Park"}
```

RESULTS 1-1 OF 1

```
_id: ObjectId("573a1399f29313caabcedc5d")
fullplot: "Huge advancements in scientific technology have enabled a mogul to cre...""
> imdb: Object
year: 1993
plot: "During a preview tour, a theme park suffers a major power breakdown th..."
> genres: Array
rated: "PG-13"
metacritic: 68
title: "Jurassic Park"
lastupdated: "2015-08-31 00:04:50.280000000"
> languages: Array
> writers: Array
type: "movie"
> tomatoes: Object
poster: "https://m.media-amazon.com/images/M/MV5BMjM2MDgxMDg0Nl5BMl5BanBnXkFtZT...)"
num_mflix_comments: 402
released: 1993-06-11T00:00:00.000+00:00
> awards: Object
> countries: Array
> cast: Array
> directors: Array
runtime: 127
```

# In MongoDB

---

The cast is simply given as an *array*:

```
num_mflix_comments: 402
released: 1993-06-11T00:00:00.000+00:00
> awards: Object
> countries: Array
< cast: Array
  0: "Sam Neill"
  1: "Laura Dern"
  2: "Jeff Goldblum"
  3: "Richard Attenborough"
> directors: Array
runtime: 127
```

# Organisation I

---

How do you organise your stuff — say, your digital photos?

- ▶ Carefully create folders for each year 2018, 2019, 2020 etc.
- ▶ Within each *year-folder* create *subject folders* Family, School, Work, Friends, etc.
- ▶ Create additional folders for various criteria

In this hierarchical organization every photo has a precise place.

## Organisation II

---

How do you organise your digital photos?

- ▶ Have one giant folder containing all your photos
- ▶ Use *EXIF metadata* and of *tags* to find specific photos

*Organisation* (SQL) is replaced by *search* (NoSQL).

## Scaling out

---

The easiest way to deal with larger datasets and/or more queries is simply to add more servers — this is called *scaling out* or *horizontal scaling*.

With MongoDB, the data can very easily be divided among as many servers as needed — just divide the collections, or even the individual documents between the servers.

If a server has the right document, then it has everything it needs.

## Scaling out in an RDBMS

---

For RDBMS, scaling out is far harder.

The format in which data is *organised* is not the format in which data is *used*.

Tables cannot be easily distributed across several servers, because queries involve accessing multiple tables.

CITS1402  
Relational Database Management Systems

Seminar 1

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Databases

---

What is the purpose of a database?

A DBMS provides *“efficient, reliable, convenient, and safe multi-user storage of – and access to – massive amounts of persistent data.”* (Jennifer Widom, Stanford).

# Relational Databases

---

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.

# Teaching Week 1

---

- ▶ Video 01 Introduction
- ▶ Video 02 Unit Details
- ▶ Video 03 The Relational Model
- ▶ Video 04 SQL1

# Locating Resources

---

- ▶ Echo  
Lecture videos and annotated PDF slides are on Echo360.
- ▶ The LMS  
Weekly worksheets, additional notes, database files, unit outline, various links
- ▶ help1402  
Help forum for posting and answering questions that are not personal
- ▶ cits1402-pmc@uwa.edu.au  
Questions with a personal or private component

# Video 01 Introduction

---

- ▶ The need to store data
- ▶ The basic CRUD operations  
*Create, Read, Update and Delete*
- ▶ Dominance of the relational model

## Video 02 Unit Details

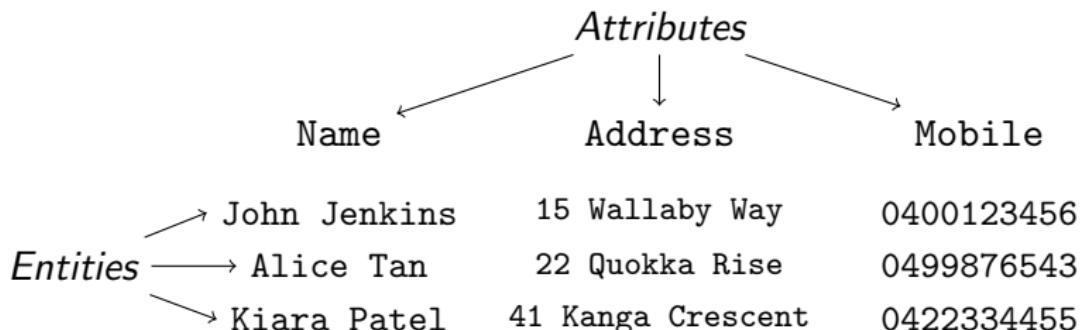
---

What you actually do week-to-week.

# Video 03 The Relational Model

---

All data is stored in *rectangular tables*.



# Tabular Terminology

---

- ▶ Table contains details about an *entity set*
- ▶ Each *instance* or *entity* is represented by one *row* or *tuple*
- ▶ Each *column* stores the values of a single *attribute* or *field*

A table represents an *entity set* or, in later weeks, a *relationship* between two or more entity sets.

(\*) A table is in *first normal form* 1NF if each attribute is *single-valued*.

## Video 4 SQL1

---

Structured Query Language (SQL) is used to:

- ▶ Create empty tables (`CREATE TABLE`)
- ▶ Insert data into tables (`INSERT INTO`)
- ▶ Read data from a table or tables (`SELECT`)
- ▶ Delete data (`DELETE`) or entire tables (`DROP`)

Most of SQL deals with *reading information* from a database, and so SQL “programs” are called *queries*.

# SQLite

---

Many *implementations* of SQL, but we will be using SQLite.

- ▶ A database is a single file on disk
- ▶ Access the database by
  - ▶ Interactive command line program `sqlite3`
  - ▶ A graphical user interface `SQLiteStudio`

Downloads and documentation are located at [sqlite.org](http://sqlite.org).

# Structure of a query

---

Starts with the word **SELECT**

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

and ends with a semicolon.

## Start off simple

---

```
SELECT <columns>
FROM <tables>
WHERE <rowconditions>;
```

## The critical concept

---

The `SELECT` statement works with *one row at a time* in the following manner:

- ▶ The `FROM` part specifies where to find the input rows
- ▶ The `WHERE` part filters out undesirable rows
- ▶ The `SELECT` part constructs the output rows

SQL applies the same mechanical process to each row in turn.

In particular, the basic `SELECT` statement *cannot compare* two different rows.

# AFLResult

---

The first lab will use a database AFLResult.db.

year	round	homeTeam	awayTeam	homeScore	awayScore
2012	1	GWS	Sydney	37	100
2012	1	Richmond	Carlton	81	125
..					
..					

*I frequently use sports data, because it is the best source of real data that is clean, unambiguous and intuitively easy to understand. No special knowledge of, or interest in, the actual sport concerned (or any sport at all) is required.*

## Start off simple

---

```
SELECT homeTeam, awayTeam  
FROM AFLResult  
WHERE homeScore > 150;
```

## Saving queries in files

---

Next week's lab asks you to create *individual files* called A1.sql, A2.sql and A3.sql.

These files should contain *one SQL query* that starts with SELECT and terminates with a semi-colon.

Three golden rules for preparing these files

1. Use a *plain text editor*, such asTextEdit or Notepad++
2. Make sure files are *saved* in plain text format (not RTF)
3. (Windows) *Turn off* file-name extension hiding

***Never*** use MS Word for SQL code.

## Some gotchas

---

A *gotcha* is a “sudden unforeseen problem”, and there are plenty of places where this happens with computers / SQL / SQLite.

We'll gradually build up a repertoire of gotchas as semester progresses.

## Gotcha 1 — Lost in the file system

---

When Terminal/Powershell is being used, it keeps track of a location in the file system, usually called the *working directory*.

Any file names used for reading or writing are interpreted *relative to* the working directory.

So if you type `.open AFLResult.db` from within SQLite, it will try to locate the file *in the working directory*.

## Gotcha 2 — Don't use fancy quotes

---

*Strings* in SQLite are delimited by single or double quotes, so either

'West Coast'    "West Coast"

These are the ASCII characters 39 and 34.

There are other *fancy quote characters* that SQLite doesn't know:

‘ “ ” ’ “ ” ’

If you *copy and paste* from a *typeset document*, such as a Word document or PDF, you will probably get fancy quotes.

## Gotcha 3 — file name extensions

---

File name extensions are used to reflect the *type of file*:

Seminar01.pdf      Seminar01.docx      AFLResults.db

Changing a file's *name* does not magically change its *contents*.

By default, Windows *hides the file name extensions* from the user.

Basically, Windows assumes that you will never create or manipulate files directly, but only through applications, and so it manages the file name extensions.

For this unit, you should *turn off* this “feature”, so that you always know the full file name of all of your files.

# Demonstration of sqlite3

---

- ▶ Start command-line interface
- ▶ Open existing database
- ▶ Dot commands
- ▶ SQL queries

CITS1402  
Relational Database Management Systems

Seminar 2

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Locating Resources 1

---

- ▶ BlackBoard Learning Management System (LMS)  
<https://lms.uwa.edu.au>
- ▶ Lecture Videos and Slides are on Echo  
Access from LMS “Lecture Recordings” or direct at  
<https://echo360.net.au> or by using the Echo360 app.

## Learning Resources 2

---

- ▶ Help forum is help1402 (non-personal queries)  
Access from LMS “Jump to help1402” or direct at  
<https://secure.csse.uwa.edu.au/run/help1402>  
Michael and I clear this twice a day Monday to Friday.
- ▶ Lab submissions are made via cssubmit  
<https://secure.csse.uwa.edu.au/run/cssubmit>
- ▶ Unit email is [cits1402-pmc@uwa.edu.au](mailto:cits1402-pmc@uwa.edu.au) (personal queries)  
Michael and I monitor this Monday to Friday.

## Teaching Week 2

---

- ▶ Video 05 RDBMS
- ▶ Video 06 Normalisation
- ▶ Video 07 [JOIN](#)

## Video 05 RDBMS

---

Talks about the “Big Picture”

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

# Data Independence / Efficiency

---

AIM: User need only know the *logical structure* of the data.

- ▶ *Data Independence*

DBMS manages all the *low-level details*, such as *where* (filenames) and *how* (data structures) the data is stored.

- ▶ *Efficiency*

SQL queries are *declarative* statements that are analysed by the *query optimiser*, which chooses an *execution plan*.

The relational model is *expressive enough* for most data, but *constrained enough* for effective query optimisation.

# Data Integrity

---

Ideally, the data in a database should always be *correct*.

Of course, this is an impossible goal, not least because human users can always enter incorrect values (usually by mistake).

The best that we can hope for is that the database always remains *internally consistent*.

In other words, the database should never contain *incompatible / contradictory* data, and preferably not *incomplete* data.

## Constraints

---

This is so important that implementations of SQL provide numerous different mechanisms to protect data integrity.

Collectively, these are called *integrity constraints* — SQL will *prevent execution* of commands that *violate* the constraints.

# Constraints

---

These constraints are *specified* when a table is defined, and then enforced by the system as the database is used.

- ▶ *Domain Constraints*

Add `CHECK` to a column definition to indicate its *domain*.

- ▶ *Key Constraints*

Specify the *primary key*, which is the value or values that uniquely determine an entity.

- ▶ *Referential Integrity*

Specify when a column of Table A *refers to* a column of Table B, so SQL can ensure that all cross-table references are correct.

## Video 06 Normalisation

---

Poor database design can lead to potential data integrity problems that are impossible for SQL to prevent.

In particular, *redundancy* in a database, where data is stored in more than one table, can lead to various *anomalies*.

For example, if a student with number 1001 scored 82 in CITS1402 in 2020, then this *association*

snum	year	unit	mark
1001	2020	CITS1402	82

should not occur in *more than one table*.

## Normalisation

---

When a database is first designed, the designer must decide what *tables* are needed, and what *columns* each table should contain.

There is an extensive theory of *normal forms* whereby the designer can eliminate certain types of redundancy (and other issues) by ensuring that the database structure obeys certain constraints.

The basic question in *normalisation* is whether or not it is worthwhile to split one table into multiple tables.

The most common normal forms are 1NF, 2NF, 3NF and BCNF.

## Video 07 JOIN

---

Suppose the CSSE department keeps records of its students in the following form

num	name	email	unitCode	mark
1001	Amy	amy@gmail	CITS1402	82
1001	Amy	amy@gmail	CITS2402	77
1001	Amy	amy@gmail	CITS1401	85
1002	Bai	bai@qq.com	CITS1402	62
1002	Bai	bai@qq.com	CITS2402	88
1003	...			

Here there is obvious redundancy, leading to possible anomalies.

# Split the tables

---

Two tables — **Student** and **Mark**:

num	name	email
1001	Amy	amy@gmail
1002	Bai	bai@qq.com
1003	...	

num	unitCode	mark
1001	CITS1402	82
1001	CITS2402	77
1001	CITS1401	85
1002	CITS1402	62
1002	CITS2402	88
1003	...	

## Join together

---

Send congratulatory email to every student with more than 80%.

We'd like to have the original "redundant" table back again!

SQL can *join* the two tables "on the fly" as and when needed.

SQL forms the joined table by pairing up a row of **Student** and a row of **Mark**.

# The Cartesian product

---

```
sqlite> .headers on
sqlite> .mode column
sqlite> .width 4 3 13 4 8 4
sqlite> SELECT * FROM Student JOIN Mark;
num    nam   email           num    unitCode  mark
----  ----  -----  ----  -----  -----
1001  Amy   amy@gmail.com  1001  CITS1402  82
1001  Amy   amy@gmail.com  1001  CITS2402  77
1001  Amy   amy@gmail.com  1001  CITS1401  85
1001  Amy   amy@gmail.com  1002  CITS1402  62
1001  Amy   amy@gmail.com  1002  CITS2402  88
1002  Bai   bai@qq.com   1001  CITS1402  82
1002  Bai   bai@qq.com   1001  CITS2402  77
1002  Bai   bai@qq.com   1001  CITS1401  85
1002  Bai   bai@qq.com   1002  CITS1402  62
1002  Bai   bai@qq.com   1002  CITS2402  88
```

## The JOIN condition

---

```
SELECT *
FROM Student JOIN Mark USING (num);
```

```
sqlite> SELECT *
...> FROM Student JOIN Mark USING (num);
num      nam    email           unit   mark
----  -----  -----
1001    Amy    amy@gmail.com    CITS    85
1001    Amy    amy@gmail.com    CITS    82
1001    Amy    amy@gmail.com    CITS    77
1002    Bai    bai@qq.com     CITS    62
1002    Bai    bai@qq.com     CITS    88
```

## Now use it

---

```
SELECT unitCode, email  
FROM Student JOIN Mark USING (num)  
WHERE mark >= 80;
```

```
sqlite> .mode list  
sqlite> SELECT unitCode, email  
...> FROM Student JOIN Mark USING (num)  
...> WHERE mark >= 80;  
unitCode|email  
CITS1402|amy@gmail.com  
CITS1401|amy@gmail.com  
CITS2402|bai@qq.com
```

## The take-home message

---

- ▶ *Data Integrity* is enhanced by eliminating *redundancy*.
- ▶ This tends to require data split into *many small tables*
- ▶ *Using* the database relies on *joining* multiple small tables.

Logically speaking, a `JOIN` is equivalent to forming the full Cartesian product, then applying the `JOIN` condition.

Of course, the query optimiser devises a much better execution plan than this!

## Gotcha 4 — SQL is not English

---

Here is a query in English: “*List the details for all matches where the home team is Hawthorn or Fremantle.*”

```
SELECT * FROM AFLResult  
WHERE homeTeam == 'Hawthorn' OR 'Fremantle';
```

But the syntax of `OR` is

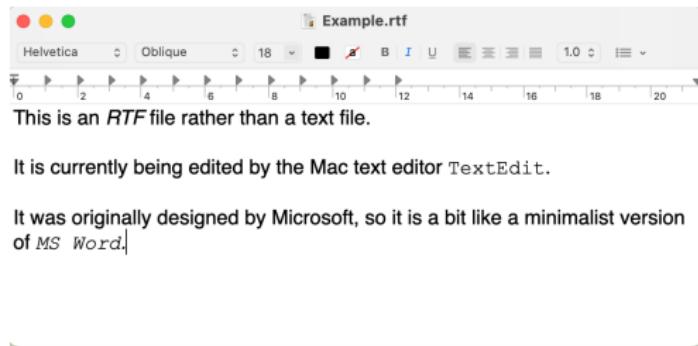
```
<boolean_expression1> OR <boolean_expression2>  
homeTeam == 'Hawthorn' OR 'Fremantle'
```

What is '`Fremantle`' as a boolean expression?

## Gotcha 5 — Avoid RTF files

---

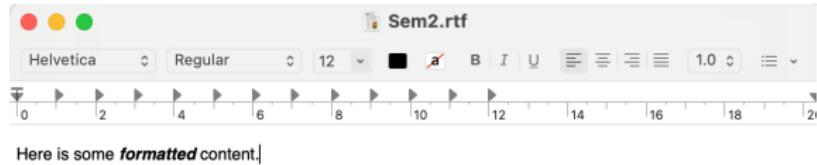
Rich Text Format is a special file format for combining text and simple formatting instructions, such as **bold**, *italic* and underline.



But what is *inside* an RTF file?

# Look inside

---



It is mostly *formatting commands*, not the actual text.

```
00013890@DEP52010 CITS1402_2022_Seminars % cat Sem2.rtf
{\rtf1\ansi\ansicpg1252\cocoartf2639
\cocoatextscaling0\cocoaplatform0{\fonttbl\f0\fswiss\fcharset0 Helvetica;\f1\fsw
iss\fcharset0 Helvetica-BoldOblique;}
{\colortbl;\red255\green255\blue255;}
{\*\expandedcolortbl;};
\paperw11900\paperh16840\margl1440\margr1440\vieww11520\viewh8400\viewkind0
\pard\tx566\tx1133\tx1700\tx2267\tx2834\tx3401\tx3968\tx4535\tx5102\tx5669\tx623
6\tx6803\pardirnatural\partightenfactor0

\f0\fs24 \cf0 \
Here is some
\f1\i\b formatted
\f0\i0\b0 content.}
```

CITS1402  
Relational Database Management Systems

Seminar 3

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## Reminder 1

---

To get assistance, please use:

- ▶ [help1402](#) for questions that do not involve personal information or actual code for an assessed question
- ▶ [cits1402-pmc@uwa.edu.au](mailto:cits1402-pmc@uwa.edu.au) for questions that do involve personal information

We want all questions and responses to be in a single location that both Michael and I can access, so that either of us can see the entire conversational thread at any stage.

I answer *all outstanding questions* both morning and evening on weekdays (only), while Michael often answers at other times.

## Reminder 2

---

When using help1402,

- ▶ Do not post code for an assessed question
- ▶ Make at least some effort to check previous posts
- ▶ Keep on top of what is said in the Seminar

We'll often answer repeat questions just by providing a link to the original thread or answer.

## Reminder 3

---

Be sensible and take assessments seriously.

- ▶ Start the week's lab sheet *early* in the week
- ▶ Use your lab time to *complete* the lab sheet
- ▶ Remember that *guaranteed assistance* ends 5pm Friday

## Taking responsibility

---

Imagine that I am a large business seeking a software developer, and that applicants are to submit samples of their work.

The lab sheet gives *specifications* in English for certain queries that you are to develop in SQLite. There are optional sessions available with company employees to clarify the specifications.

Your job is to *develop and test* these queries using the sample database and deliver them by the deadline in the requested format.

If your sample work is late, incorrectly formatted, or does not work, *you will not get the job.*

## Teaching Week 3

---

- ▶ Video 08 Data Definition Language
- ▶ Video 09 `SELECT`
- ▶ Video 10 Types

## Video 08 Data Definition Language

---

SQL commands are often divided into two categories, namely the Data **Definition** Language (DDL) and the Data **Manipulation** Language (DML).

- ▶ DDL commands manipulate *tables in a database*  
Commands to *create* a table, to *delete* a table, and to *alter* a table by renaming it, or by adding, deleting or renaming columns.
- ▶ DML commands manipulate *rows in a table*  
Commands to insert rows into a table, delete rows from a table or alter the values stored in rows of a table.

## Quick View

---

These are the main keywords in each category:

DDL	DML
CREATE	SELECT
ALTER	INSERT
RENAME	UPDATE
DROP	DELETE

If it affects the *data*, it is a DML command, if it affects the *structure* it is a DDL command.

# DDL for CITS1402

---

From our perspective, `CREATE TABLE` is by far the most important DDL command with `ALTER TABLE` a distant second.

The `CREATE TABLE` statement gives the *names* and *types* of the columns.

```
CREATE TABLE Student (
    num INTEGER,
    name TEXT,
    email TEXT);
```

## Altering tables

---

```
ALTER TABLE <table_name>
ADD COLUMN <col_name> <col_type>;
```

```
ALTER TABLE <table_name>
DROP COLUMN <col_name>;
```

(`ALTER TABLE DROP COLUMN` first implemented in SQLite 3.35)

## Video 09 SELECT

---

SQL queries can get complicated and can be frustrating to write.

To debug your queries, mentally run through *exactly* what SQL is doing when it runs a query.

```
SELECT <column_expressions>
FROM <table_expression>
WHERE <boolean_expression>;
```

In practice

- ▶ `FROM <table_expression>` creates a table
- ▶ `WHERE <boolean_expression>` filters out (entire) rows
- ▶ `SELECT <column_expressions>` constructs output rows

## An aside about WHERE

---

WHERE has a *boolean expression* that is *evaluated* for each row.

Only the rows that *satisfy* the boolean condition (i.e, where the boolean expression is true) go to the next stage.

- ▶ `homeTeam == 'Fremantle'`
- ▶ `homeScore = awayScore + 1`
- ▶ `homeScore - awayScore > 20 OR awayTeam != 'Fremantle'`
- ▶ `homeTeam`

## Building the output row

---

The `SELECT` receives rows from `WHERE` filter, *one at a time*.

For each of these “input rows”, it constructs an “output row” according to a specification involving one or more of

- ▶ Column names
- ▶ Literals (numbers or strings)
- ▶ Expressions using column names and literals
- ▶ Functions

In addition, all of these output columns can be *renamed*, and `*` is shorthand for “all input columns”.

## Fremantle's home wins

---

```
SELECT year, homeTeam, awayTeam
FROM AFLResult
WHERE homeTeam == 'Fremantle'
AND homeScore > awayScore;
```

## Video 10 Types

---

SQLite is significantly different to other implementations of SQL, which usually have many different *data types*, giving the user *fine-grained control* over precise storage requirements.

```
CREATE TABLE Student (
    sNum CHAR(8),
    sName VARCHAR(64),
    sAddress VARCHAR(256),
    gender ENUM('M','F','X'),
    guildMember BOOLEAN
);
```

This *static typing* can be viewed as having a *fixed-size container* to store each value.

## In contrast

---

- ▶ SQLite has *very few* data types  
Just INTEGER, REAL, TEXT and BLOB (and NULL)
- ▶ SQLite is *dynamically typed*  
SQLite *automatically adjusts* the size of the container according to the value being stored.
- ▶ SQLite is very *permissive*  
SQLite doesn't care if you mix types, even if it *doesn't strictly make sense*.

```
sqlite> SELECT '123' + 17;
```

140

## Dealing with types

---

How should *you* approach learning about data types if SQLite doesn't care about them?

Just use the SQLite types for this unit, as this will make transitioning to other implementations of SQL straightforward.

In particular,

- ▶ Important to develop the habit of thinking about data types.
- ▶ The SQLite types are the most fundamental types in RDBMS.
- ▶ Code using SQLite types will work in other versions of SQL.

## Gotcha 7 — Avoid passivity

---

Every semester, I get messages saying

- ▶ “Is my query returning the right answer”?
- ▶ “I tried all these things, but my query doesn’t work”

You can (and should) *test your own* queries to confirm that the output makes sense.

*Randomly permuting* SQL keywords and hoping for the best is not a programming technique.

Be *proactive* in *writing*, *debugging* and *testing* queries, not passive.

CITS1402  
Relational Database Management Systems

Seminar 4

Gordon Royle

Department of Mathematics & Statistics



# SQL

---

- ▶ SQL is an ISO *standard* that specifies the *syntax* and *semantics* of a *hypothetical* database language
- ▶ SQLite 3 is an *actual* database language that is an approximation to SQL
- ▶ sqlite3 is a *command line interface* (CLI) for SQLite 3
- ▶ SQLite Studio is a *graphical user interface* (GUI) for SQLite 3

You need to *use* one or both of sqlite3 and SQLite Studio to create queries that run, but it is only SQL and SQLite that are assessed.

## The next step

---

```
SELECT <aggregate_expressions>
FROM <table_expression>
WHERE <boolean_expression>
GROUP BY <column_names>
```

### In practice

- ▶ `FROM <table_expression>` creates a table
- ▶ `WHERE <boolean_expression>` filters out (entire) rows
- ▶ `GROUP BY <column_names>` forms rows into groups
- ▶ `SELECT <aggregate_expressions>` builds *one summary row per group*

# Go Eagles!

---

How many *wins* have West Coast had in *each year* from 2012.

- ▶ Use `AFLResult` as the “input table”
- ▶ Filter out any games that are not West Coast wins
- ▶ Form *groups of rows* based on the value in the `year` column
- ▶ From each group of rows, form a *summary row* by counting the number of rows in the group

## The query

---

```
SELECT year, COUNT( * )
FROM AFLResult
WHERE (homeTeam = 'West Coast' AND
       homeScore > awayScore) OR
       (awayTeam = 'West Coast' AND
       awayScore > homeScore)
GROUP BY year;
```

# The output

---

```
1 SELECT year, COUNT( * )
2 FROM AFLResult
3 WHERE (homeTeam = 'West Coast' AND
4         homeScore > awayScore) OR
5         (awayTeam = 'West Coast' AND
6         awayScore > homeScore)
7 GROUP BY year;
```

Grid view

Total rows loaded: 10

	year	COUNT(*)
1	2012	15
2	2013	9
3	2014	11
4	2015	16
5	2016	16
6	2017	12
7	2018	16
8	2019	15
9	2020	12
10	2021	10

# The process

---

The `FROM` and `WHERE` clauses create a table listing *only* the games won by West Coast.

```
sqlite> SELECT *
...>   FROM AFLResult
...> WHERE (homeTeam = 'West Coast' AND
...>           homeScore > awayScore) OR
...>           (awayTeam = 'West Coast' AND
...>           awayScore > homeScore);
2012|1|Western Bulldogs|West Coast|87|136
2012|2|West Coast|Melbourne|166|58
2012|3|GWS|West Coast|69|150
...
...
2021|13|West Coast|Richmond|85|81
2021|18|Adelaide|West Coast|56|98
2021|19|West Coast|St Kilda|94|86
```

## The rows are grouped

---

The **GROUP BY** year means “Form the rows into groups that have the same value of year”

The 2012 group:

```
2012|1|Western Bulldogs|West Coast|87|136
2012|2|West Coast|Melbourne|166|58
...
2012|21|Port Adelaide|West Coast|50|98
2012|22|West Coast|Collingwood|107|58
```

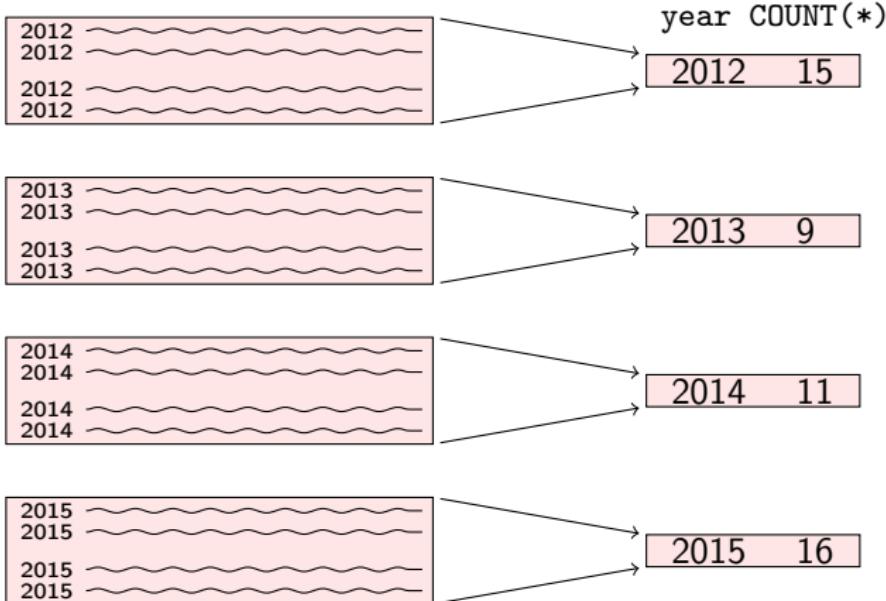
The 2013 group

```
2013|3|Melbourne|West Coast|83|177
2013|6|West Coast|Western Bulldogs|137|67
...
2013|19|West Coast|Gold Coast|130|113
2013|20|Essendon|West Coast|67|120
```

## Summary Row

---

Then `SELECT` describes *one summary row* for each *group*.



## The groups are summarised

---

The `SELECT` statement should use *only*

- ▶ Columns named in the `GROUP BY` clause, or

For example, the column `year` makes sense, because it is a *group property*, while `round` doesn't, because different rows in the same group have different values.

- ▶ Aggregate or summary functions

An aggregate or summary function, like `COUNT` is *designed* to summarise a collection of rows producing a single value

A column in `SELECT` that is not in `GROUP BY` is called a *bare column*.

# Avoid bare columns!

---



What happens if you run a query that involves a bare column?

A *cautious* language would *refuse to run* the query, or at least give a warning about the bare column.

However, like most other implementations of SQL, SQLite assumes that the user *knows what they are doing* and uses the value of the bare column from the first row of the group.

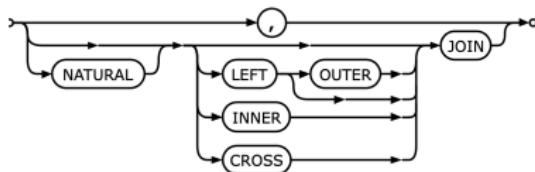
If the user *happens to know* that the values in the bare column are the same for each row in the same group, then this is ok.



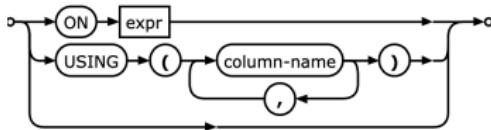
# Ways to JOIN

---

## The join-operator



## The join-clause



**FROM** combines a *join operator* with a *join clause*.

## How they work

---

Here are the “compact” schemas for `people` and `castmembers`

```
people(person_id, name, born, died)  
castmembers(title_id, person_id, characters)
```

In `imdb_top_250.db` there are 1934 rows in `people` and 1018 rows in `castmembers`.

The full *Cartesian product* forms rows by adjoining *every row* from `people` with every row from `castmembers`.

# Cartesian Product Example

---

sqlite> SELECT * FROM people, castmembers LIMIT 5;							
person_id	name	born	died	title_id	person_id	characters	
nm0000005	Ingmar Bergman	1918	2007	tt0012349	nm0088471	['His Assistant']	
nm0000005	Ingmar Bergman	1918	2007	tt0012349	nm0000122	['A Tramp']	
nm0000005	Ingmar Bergman	1918	2007	tt0012349	nm0701012	['The Woman']	
nm0000005	Ingmar Bergman	1918	2007	tt0012349	nm0001067	['The Child']	
nm0000005	Ingmar Bergman	1918	2007	tt0012349	nm0588033	['The Man']	

The Cartesian product has  $1934 \times 1018 = 1968812$  rows.

In *relational algebra* this is the table

people  $\times$  castmembers

## Other ways

---

All of these will produce the same Cartesian product.

```
SELECT * FROM people, castmembers;
SELECT * FROM people JOIN castmembers;
SELECT * FROM people INNER JOIN castmembers;
SELECT * FROM people CROSS JOIN castmembers;
```

To extract the rows we want, we need to add a `JOIN` clause.

# Ways to join I

---

```
SELECT *
FROM people JOIN castmembers
ON people.person_id = castmembers.person_id;
```

```
sqlite> SELECT *
...> FROM people JOIN castmembers
...> ON people.person_id = castmembers.person_id
...> LIMIT 5;
+-----+-----+-----+-----+-----+-----+-----+
| person_id | name | born | died | title_id | person_id | characters |
+-----+-----+-----+-----+-----+-----+-----+
| nm0088471 | B.F. Blinn | 1872 | 1941 | tt0012349 | nm0088471 | ['His Assistant'] |
| nm0000122 | Charles Ch | 1889 | 1977 | tt0012349 | nm0000122 | ['A Tramp'] |
| nm0701012 | Edna Purvi | 1895 | 1958 | tt0012349 | nm0701012 | ['The Woman'] |
| nm0001067 | Jackie Coo | 1914 | 1984 | tt0012349 | nm0001067 | ['The Child'] |
| nm0588033 | Carl Mille | 1894 | 1979 | tt0012349 | nm0588033 | ['The Man'] |
```

This table has 1018 rows, one for each row in `castmembers`.

## Ways to join II

---

```
SELECT *
FROM people JOIN castmembers
USING (person_id);
```

```
sqlite> SELECT *
...> FROM people JOIN castmembers
...> USING (person_id)
...> LIMIT 5;
person_id      name        born      died      title_id    characters
-----+-----+-----+-----+-----+-----+
nm0088471     B.F. Blinn   1872     1941     tt0012349  ['His Assistant']
nm0000122     Charles Ch   1889     1977     tt0012349  ['A Tramp']
nm0701012     Edna Purvi  1895     1958     tt0012349  ['The Woman']
nm0001067     Jackie Coo   1914     1984     tt0012349  ['The Child']
nm0588033     Carl Mille   1894     1979     tt0012349  ['The Man']
```

This has *deleted* the duplicate `person_id` column.

## Ways to join III

---

It is very common to want to join two tables on *all columns* that have the *same name*.

```
SELECT *
FROM people NATURAL JOIN castmembers;
```

```
sqlite> SELECT *
...> FROM people NATURAL JOIN castmembers;
person_id    name      born      died      title_id   characters
-----  -----
nm0088471    B.F. Blinn  1872     1941     tt0012349  ['His Assistant']
nm0000122    Charles Ch  1889     1977     tt0012349  ['A Tramp']
nm0701012    Edna Purvi 1895     1958     tt0012349  ['The Woman']
nm0001067    Jackie Coo  1914     1984     tt0012349  ['The Child']
nm0588033    Carl Mille  1894     1979     tt0012349  ['The Man']
```

This joins rows from the two tables only if they have the same values on *all columns* with the same name, then removes one of the two copies of each duplicated column.

## Looking ahead

---

In “Operators”, the video mentions `IN` and `NOT IN`.

These operators are used to check whether a value is contained (or not contained) in a *set of values*.

```
SELECT *
  FROM AFLResult
 WHERE (homeTeam == 'Fremantle' OR
        homeTeam == 'West Coast') AND
        (awayTeam == 'Port Adelaide' OR
        awayTeam = 'Adelaide');
```

## Looking ahead II

---

A *set of values* can be created using a comma-separated list of values inside normal brackets.

```
SELECT *
FROM AFLResult
WHERE
    homeTeam IN ('West Coast', 'Fremantle')
        AND
    awayTeam IN ('Port Adelaide', 'Adelaide');
```

Using these is much slicker and more natural than long *disjunctions* (boolean expressions connected by **OR** ).

CITS1402  
Relational Database Management Systems

Seminar 5

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## Writing queries

---

When dealing with SQL, always remember that *you are in charge*.

When devising a query, *be systematic*, and start by asking

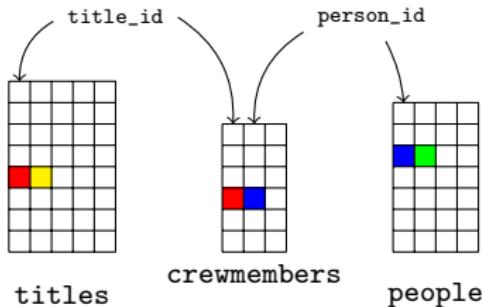
- ▶ What data is required?
- ▶ Which table(s) *contain* the data?
- ▶ How can I *join* these tables?

Make sure you *think through* the query carefully, and *write it down* before you start typing into the computer.

## List titles and screenwriters

---

- ▶ The table `titles` contains movie titles
- ▶ The table `people` contains people's names
- ▶ The table `crewmembers` identifies screenwriters



So somehow, somewhere, we must ensure that

```
titles.title_id == crewmembers.title_id  
crewmembers.person_id == people.person_id
```

## Now debug it

---

When you finally run the query, it may *work first time* 😊, or:

- ▶ You get a *syntax error*

```
sqlite> SELECT * FROM titles WHERE name LIKE "%Ford%";  
Error: in prepare, no such column: name (1)
```

SQLite, like most other implementations of SQL has terse and cryptic error messages.

- ▶ The query runs but produces *incorrect output*

## Example

---

Write a single SQL query that lists the titles of all the IMDB Top-250 movies with more than 2 million votes.

The screenshot shows a database interface with an SQL editor and a results grid. The SQL query is:

```
1 SELECT title
2 FROM titles JOIN ratings
3 WHERE votes > 2000000;
4 |
```

The results grid displays the following data:

	title
1	12 Angry Men
2	12 Years a Slave
3	1917
4	2001: A Space Odyssey
5	3 Idiots
6	8½
7	A Beautiful Mind
8	A Clockwork Orange

Total rows loaded: 1500

## A three-step process

---

Writing a query for submission is a three-step process:

- ▶ *Designing* the query
- ▶ *Implementing* and *debugging* the query
- ▶ *Testing* the query

## Testing the query

---

To test the query, take a *critical look* at the *output* of the query.

Using your ability to examine the data in the database (using sqlite3 or SQLStudio):

- ▶ (Correctness) Check some of the rows that your query *has produced*, and make sure that they are correct.
- ▶ (Completeness) Work out some rows that *should* be in the output, and check that they actually *are* in the output.

(Ensuring that a system is *correct* and *complete* is a recurring theme in CS.)

# Relational Algebra

---

Normal algebra is the symbolic manipulation of *arithmetic expressions*

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

where variables like  $x, y$  represent *numbers*.

Relational algebra is the symbolic manipulation of *relational expressions*, where variables like  $R, S$  represent *relations*.

A relational expression is a precise specification of a desired *output relation* in terms of known relations.

## Relational Variables

---

A relational is a *set of tuples*—almost the same as a *table*, but with no repeated rows.

A	B	C
1	1	3
2	1	4
1	1	2

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

# The relational operators

---

The main relational operators are

- ▶ The *projection operator*  $\pi$
- ▶ The *selection operator*  $\sigma$
- ▶ The *Cartesian product operator*  $\times$
- ▶ The *join operator*  $\bowtie$

Unfortunately, there is a terminology clash between SQL and relational algebra.

The word `SELECT` in SQL does not correspond to *selection* in relational algebra.

# Projection

---

The *projection operator*  $\pi$  specifies which *columns* to keep. So

$$\pi_{A,B}(R)$$

means to keep columns  $A$  and  $B$  only.

A	B	C
1	1	3
2	1	4
1	1	2

Duplicates are removed so

$$\pi_{A,B}(R) = \{(1, 1), (2, 1)\}.$$

## Selection

---

The *selection operator*  $\sigma$  determines which *rows* to keep. So

$$\sigma_{A=B}(R)$$

means to keep all rows where  $A = B$ .

A	B	C
1	1	3
2	1	4
1	1	2

$$\sigma_{A=B}(R) = \{(1, 1, 3), (1, 1, 2)\}.$$

# Products

---

The *Cartesian product*

$$R \times S$$

means to form every possible tuple obtained by “gluing together” a tuple of  $R$  and a tuple of  $S$ .

If  $R$  has  $r$  tuples, and  $S$  has  $s$  tuples, then  $R \times S$  has  $rs$  tuples.

## Example

---

With  $R$  as above, what is

$$\pi_{A,B}(R) \times \sigma_{A=B}(R)?$$

$$\begin{array}{r} \overline{\begin{array}{cc} A & B \end{array}} \\ \begin{array}{cc} 1 & 1 \\ 2 & 1 \end{array} \end{array} \times \begin{array}{r} \overline{\begin{array}{ccc} A & B & C \end{array}} \\ \begin{array}{ccc} 1 & 1 & 3 \\ 1 & 1 & 2 \end{array} \end{array} = \begin{array}{r} \overline{\begin{array}{ccccc} A & B & A & B & C \end{array}} \\ \begin{array}{ccccc} 1 & 1 & 1 & 1 & 3 \\ 1 & 1 & 1 & 1 & 2 \\ 2 & 1 & 1 & 1 & 3 \\ 2 & 1 & 1 & 1 & 2 \end{array} \end{array}$$

## The join operator

---

We'll meet the join operator  $\bowtie$  in a future video, and while its behaviour is slightly more complicated than the others, it is not difficult to understand.

Evaluating a relational expression is not difficult provided you are *methodical*.

In this unit, we really only use the *notation* of relational algebra (rather than manipulating relational expressions).

# Classic Models

---

The database `classicmodels.db` is an SQLite database derived from an old widely-used teaching database created for MySQL.

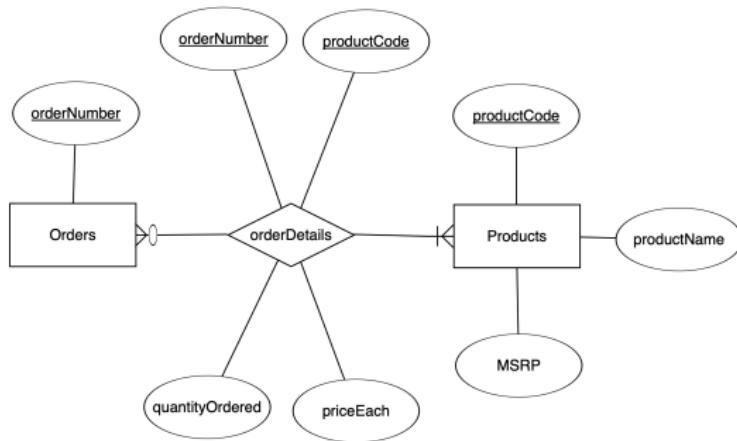
It has the following tables:

- ▶ `customers`
- ▶ `employees`
- ▶ `offices`
- ▶ `orderdetails`
- ▶ `orders`
- ▶ `payments`
- ▶ `productlines`
- ▶ `products`

# The tables

---

Mostly, we will use the tables `orders`, `orderdetails` and `products`, and even then we'll ignore most of the fields.



## How it works

---

An *order* has a unique *order number* and is associated with a particular *customer*.

The actual items that make up an individual order are stored in the rows of *orderdetails*.

Each row of *orderdetails* contains information about *one product*, in particular the *productCode*, the *quantityOrdered* and the *priceEach*.

# Order 10200

---

Query History

```
1 SELECT * FROM orderDetails WHERE orderNumber = 10200;
2 |
```

Grid view Form view

Total rows loaded: 6

	orderNumber	productCode	quantityOrdered	priceEach	orderLineNumber	
1	10200	S18_2581	28	74.34	3	
2	10200	S24_1785	33	99.57	5	
3	10200	S24_4278	39	70.28	4	
4	10200	S32_1374	35	80.91	1	
5	10200	S32_4289	27	65.35	6	
6	10200	S700_2834	39	115.09	2	

## Order 10200

---

This order contains

- ▶ 28 units of product S18\_2581 at a cost of \$74.34 per unit.
- ▶ 33 units of product ...
- ▶ ...
- ▶ 27 units of product ...
- ▶ 39 units of product S700\_2834 at a cost of \$115.09 per unit.

The total cost (to the customer) of this order is

$$28 \times 74.34 + 33 \times 99.57 + 39 \times 70.28 + 35 \times 80.91 + 27 \times 65.35 + 39 \times 115.09 = 17193.06$$

(Note that SQLite returns 17193.059999999998 for this query due to round-off errors in floating-point arithmetic.)

# Line Items

---

NAME :	Corinthian Hills temple drive		
ADDRESS:	Tel#638-0675		
TIN :			
BUSINESS STYLE:			
TRANSACTION CODE 1			
1	HEALTHPLUSCANOLA OIL 1	194.00	V
1	KNORR LIQUID SEASONI	45.25	V
1	SBSUGARWASHED1KG	52.95	V
1	MANGTOMASALLAROUNDSA	31.75	V
1	SKYFLAKES 10 SINGLE	47.50	V
2	NISSINWAFERCHOCO @53.50	107.00	V
1	J J PIATTOS CHEESE	27.00	V
2	J J V CUT BBQ 60 @24.25	48.50	V
1	CLOVER CHEESE FAMILY	27.00	V
1	CEBUBRANDDRDMANGOSLI	115.00	V
0.088	V KINCHAY KG @392.00	34.50	N
1.042	MIKISA CRAB CLAW @828.00	862.78	N
0.504	MIKISA LAPU LAPU @738.00	371.95	N
1	BOUNTY FRESH MEDIUM	92.50	N
1	SURFPOWDERPURPLEBL00	162.50	V
6	SURFPOWDERPURPLE @5.00	30.00	V
5	SBINTERLEAVEDBTR @54.10	270.50	V
1	SBBAATHROOMTISSUE1000	119.90	V
TAL			
	PESO	2,640.58	
PROD CNT: 18 TOT QTY: 27.63		2,640.58	
VAT SALE			

# Complex computing problems

---

program. A complex computing problem will normally have some or all of the following criteria:

1. involves wide-ranging or conflicting technical, computing, and other issues;
2. has no obvious solution, and requires conceptual thinking and innovative analysis to formulate suitable abstract models;
3. a solution requires the use of in-depth computing or domain knowledge and an analytical approach that is based on well-founded principles;
4. involves infrequently encountered issues;
5. is outside problems encompassed by standards and standard practice for professional computing;
6. involves diverse groups of stakeholders with widely varying needs;
7. has significant consequences in a range of contexts;
8. is a high-level problem possibly including many component parts or sub-problems;
9. identification of a requirement or the cause of a problem is ill defined or unknown. (Seoul Accord, Section D)

## Remaining Time

---

Interactively explore Classic Models.

CITS1402  
Relational Database Management Systems

Seminar 6

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# The Test

---

## Details for the test

- ▶ On LMS, from 10am Saturday 18th September
- ▶ 50 minutes
- ▶ Open book
- ▶ 8 multiple choice / short-answer questions (answer in LMS)
- ▶ 6 query writing questions (type answers into LMS)
- ▶ Questions randomly selected from “pools” of same difficulty

Previous years' tests were slightly different (no short-answer questions)

## Test Topics

---

The test may cover

- ▶ Videos 1–20 (All of Weeks 1–6 and first video from Week 7)
- ▶ All Labs from Weeks 2–7
- ▶ Seminars from Weeks 1–7
- ▶ All LMS Announcements
- ▶ All answers on `help1402`

## Sample MCQ

---

Suppose that R is a table with 5 rows and 3 columns, and that S is a table with 4 rows and 2 columns. How many rows are there in the table produced by the SQL query?

```
SELECT * FROM R, S;
```

1. 20
2. 9
3. 6
4. 26
5. 5

## MCQ2

---

Consider a table with schema

Part(partID INTEGER, supplier TEXT, price REAL)

which contains data about the prices charged for industrial parts by different suppliers. Each part is uniquely identified by partID, each supplier is uniquely identified by supplier, and the table contains no NULL entries.

What information is returned by the following query?

```
SELECT partID, COUNT(*), MIN(price)
FROM Part
GROUP BY partID
HAVING COUNT(*) > 1;
```

## MCQ2 Choices

---

1. For each part in the table, it lists the part ID, the number of suppliers of that part, and the cheapest price for that part.
2. For each part that is supplied by more than one supplier, it lists the part ID, the number of suppliers of that part, and the cheapest price for that part.
3. For each part that is supplied by more than one supplier, it lists the part ID, the name of the cheapest supplier for that part, and the cheapest price for that part.
4. For each part in the table, it lists the part ID, the number of suppliers that can supply the part at the cheapest price, and the cheapest price for that part.
5. For each supplier in the table that supplies more than one part, it lists the number of parts they supply, and the price of the cheapest one.

## MCQ3

---

Suppose that  $R(A, B)$  and  $S(B, C)$  are two tables and that currently they have the following contents:

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

How many rows are in the table produced by the query

```
SELECT *
FROM R JOIN S USING (B);
```

## MCQ3 Options

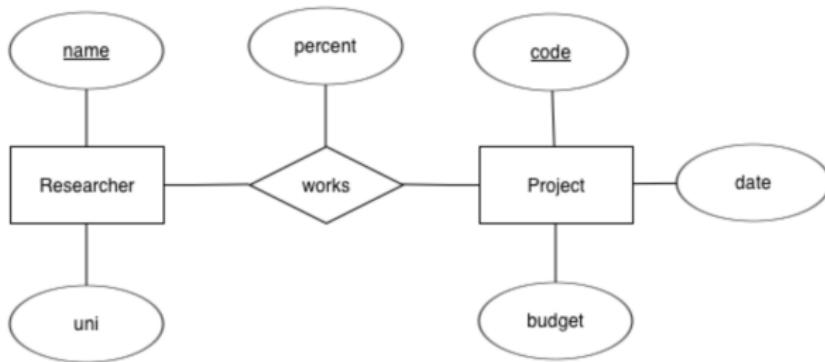
---

1. 1
2. 2
3. 3
4. 4
5. 9

# Query Writing Questions

---

Query writing questions are similar to the lab questions



## Range of questions

---

- ▶ List the codes and budgets of all projects that started in 2021.
- ▶ List the names of the researchers who work on project ARC23.
- ▶ For each project, list its *code* and the *number of researchers* working on that project
- ▶ For each project, list its *code* and the *names of the researchers* who are working on that project only.

# ERD

---

ERD expresses a database in terms of *entities* (things) and *relationships* (relations between things).

Sometimes it is easy to identify the entities / relationships, but not always, although there are many heuristics such as “nouns are entities and verbs are relationships”.

In a university, students take a unit in a particular year, and get a grade for that unit.

Demo of using ERDPlus to create a simple ERD.

You ***must use*** ERDPlus for any submitted ER Diagrams, and you ***must use*** the conventions as in this unit.

## From ERD to schema

---

An ERD is a visual representation of a database, but it has two types of thing — namely, *entities* and *relationships*.

A database has only *one* type of thing, namely *tables*, so how do we translate?

1. *Each* entity set becomes a table of the same name, whose columns are the attributes.
2. Only *some* relationships become tables (the many-to-many ones) while the others can be implemented via attributes.

## Things to watch out for

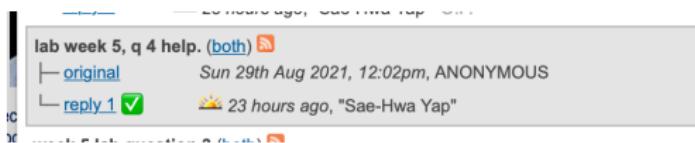
---

- ▶ Attributes on relations
- ▶ Translating relations to tables
- ▶ One-to-many and one-to-one relations

# The green tick

---

On help1402, students often answer other students' queries.



If the answer satisfactorily answers the question, then Michael and I can “verify” the answer, which labels it with a green tick.

If the student answer is more of a suggestion, a general comment, or an incomplete answer, then it normally won't be ticked.

(If the student answer is wrong or misleading, then normally I would intervene.)

CITS1402  
Relational Database Management Systems

Seminar 7

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# The Test

---

## Details for the test

- ▶ On LMS, from 10am Saturday 17th September
- ▶ 50 minutes
- ▶ Open book
- ▶ 8 multiple choice / short-answer questions (answer in LMS)
- ▶ 6 query writing questions (type answers into LMS)
- ▶ Questions randomly selected from “pools” of same difficulty

Previous years' tests were slightly different (no short-answer questions)

## Test Topics

---

The test may cover

- ▶ Videos 1–20 (All of Weeks 1–6 and first video from Week 7)
- ▶ All Labs from Weeks 2–7
- ▶ Seminars from Weeks 1–7
- ▶ All LMS Announcements
- ▶ All threads on help1402

# 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>
```

## SELECT

---

Determines the *columns* of the table that is output, using:

- ▶ Column names

```
SELECT winnerName
```

- ▶ Arithmetic expressions

```
SELECT ABS(homeScore - awayScore)
```

- ▶ Aggregate or summary functions

```
SELECT winnerName, COUNT(*)
```

Output columns can be *renamed* for convenience:

```
SELECT SUM(quantityOrdered * priceEach) AS totalCost
```

## FROM

---

The `FROM` clause constructs an initial “input table” from which the output will be constructed using:

- ▶ Table names

```
FROM ATPResult
```

- ▶ Joins of tables

```
FROM Student JOIN Enrolment USING (sNum)
```

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

```
FROM Student NATURAL JOIN Enrolment
```

```
FROM Student, Enrolment WHERE Student.sNum =  
Enrolment.sNum
```

## WHERE

---

The `WHERE` clause filters unwanted rows from the initial table using

- ▶ Simple boolean expressions using comparison operators

`WHERE homeScore > 150`

`WHERE homeTeam == 'West Coast'`

- ▶ Compound boolean expressions using boolean operators

`WHERE homeScore > 150 AND homeTeam == 'West Coast'`

`WHERE homeScore > 150 AND homeTeam != 'Fremantle'`

`WHERE (homeTeam == 'Fremantle') AND (awayTeam == 'West Coast')`

### *Warning*

`WHERE homeTeam == 'Fremantle' OR 'West Coast'`

## GROUP BY

---

Forms rows into *groups* according to the value of the specified column, or columns.

GROUP BY yr

GROUP BY sNum

GROUP BY winnerName, winnerAge

GROUP BY yr, winnerName, winnerHand

GROUP BY yr, unitCode

A *single output row* will be formed from each group according to the specifications given in the SELECT.

## GROUP BY

---

Every column specified in `SELECT` extracts *one value* from the corresponding group of rows.

- ▶ A column named in the `GROUP BY` clause
  - Every row in the group has the same value in this column
- ▶ An aggregate or summary function (`COUNT`, `AVG` etc)
  - This is a function *intended* to summarise a collection of values

Anything else is a *bare column*.

## HAVING

---

Finally, **HAVING** is a final filtering step applied to the *summary rows* that have been produced by the previous steps.

So **WHERE** filters “original rows” while **HAVING** filters “summary rows” .

```
SELECT winnerName, COUNT(*) AS numWins
FROM WTAResult
WHERE winnerCountry == 'AUS'
GROUP BY winnerName
HAVING numWins > 5;
```

## ORDER BY

---

Finally, ORDER BY arranges the output rows in order, according to the given expression.

ORDER BY yr

ORDER BY yr, numWins

ORDER BY sNum, AVG(mark)

The default order is *increasing numerical* order for numbers, and *increasing alphabetical* order for strings.

The order can be reversed by adding DESC after the field.

ORDER BY yr, numWins DESC

(This uses *ascending* year, then *descending* number of wins.)

## Subqueries

---

A *subquery* is a query that is *contained within* another query.

A subquery can be used to *dynamically incorporate values* into other queries.

A suitable subquery can be used wherever a table is given as a *named table* (or join) or a value is given as a *literal* (i.e., an actual number or string).

A subquery can also be called a *nested* query or *inner* query.

## Correlated or uncorrelated

---

An *inner query* is *uncorrelated* if it does not refer to any values from the outer query.

```
SELECT productName, quantityInStock  
FROM Products  
WHERE quantityInStock =  
(SELECT MAX(quantityInStock)  
FROM Products);
```

The inner query is independent of the outer query, and only needs to be run once.

## Useful operators

---

- ▶ The operators `IN` / `NOT IN` test whether an individual value is contained in a list of values or not.
- ▶ The operators `EXISTS` / `NOT EXISTS` check whether a subquery returns any values or not.

# Expensive Products (Uncorrelated)

---

```
SELECT *
FROM products
WHERE (productLine, buyPrice) IN
(SELECT productLine, MAX(buyPrice)
FROM products
GROUP BY productLine);
```

productLine	MAX(buyPrice)
Classic Cars	103.42
Motorcycles	91.02
Planes	77.27
Ships	82.34
Trains	67.56
Trucks and Buses	84.76
Vintage Cars	86.7

## Expensive Products (Correlated)

---

```
SELECT *
FROM products P
WHERE NOT EXISTS
(SELECT * FROM products Q
 WHERE P.productLine == Q.productLine
 AND P.buyPrice < Q.buyPrice);
```

The logic here is “Product  $P$  is the most expensive in its product line if there *does not exist* a product  $Q$  in the same product line that is *more expensive*”.

## A huge difference

---

- ▶ An *uncorrelated* subquery is run *once* and the same output *re-used* for every row processed by the outer query.
- ▶ A *correlated* subquery is (in principle) run *repeatedly*, once for each row processed by the outer query.

Most (perhaps all?) queries involving a correlated subquery can be rewritten to avoid it.

Different implementations of SQL will optimise such queries more or less effectively.

CITS1402  
Relational Database Management Systems

Seminar 8

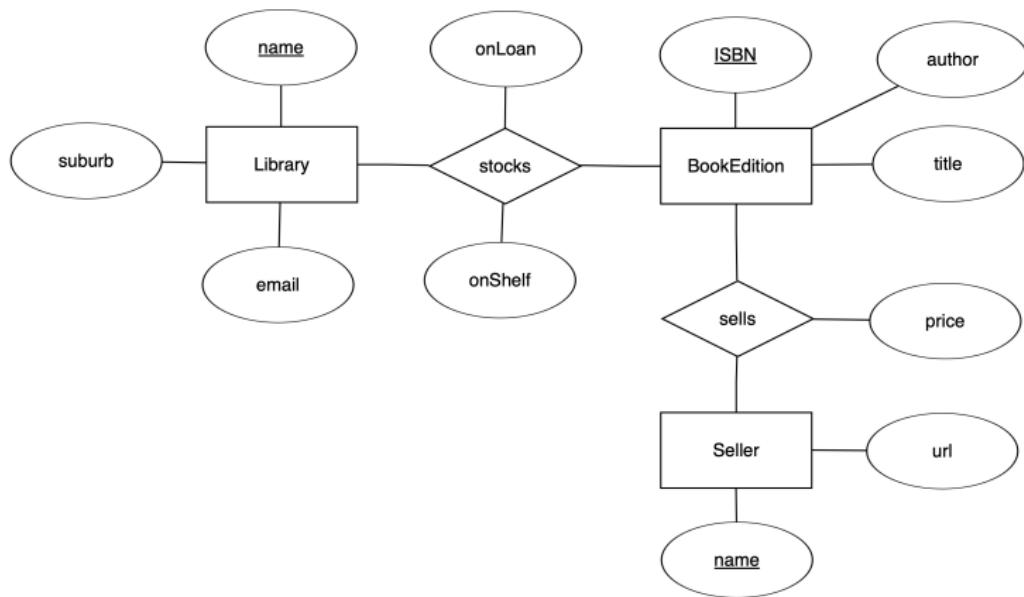
GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Library Board

---



## Question 1

---

Write a single SQL query that will list the *names* of all of the libraries that are located in either Nedlands or Subiaco, in reverse alphabetical order.

```
SELECT name  
FROM Library  
WHERE (suburb == 'Nedlands') OR (suburb == 'Subiaco')  
ORDER BY name DESC;
```

## Question 2

---

Write a single SQL query that will list the *titles*, once each, of all of the books written by Joseph Heller that are available to borrow at *The Grove Library*.

```
SELECT DISTINCT title
FROM stocks JOIN BookEdition USING (ISBN)
WHERE name = 'The Grove Library' AND author = 'Joseph
    Heller'
AND onShelf > 0;
```

What does “*available*” mean?

The sample row ('The Grove Library', '9780099529118', 2, 1) indicates that the Grove Library stocks 3 physical copies of this particular edition of *Catch 22*, of which 2 are out on loan, and 1 is on the library shelves (and hence available for borrowing).

## Question3

---

Write a single SQL query that lists, for each ISBN, the *title* of the book and the *number of libraries* that stock that book, with the output restricted to the ISBNs that are stocked by at least 3 libraries (at least 3 includes exactly 3).

```
SELECT title, COUNT(*)
FROM BookEdition JOIN stocks USING (ISBN)
GROUP BY ISBN, title
HAVING COUNT(*) > 2;
```

Note that **GROUP BY ISBN** will produce the same answer, but **title** will be a bare column.

## (Aside) Functional Dependencies

---

It is impossible to have two rows of the table with the *same* ISBN but *different* titles — this is just how ISBNs work.

So `GROUP BY ISBN` and `GROUP BY ISBN, title` produce the same groups.

In this situation we say that “ *ISBN determines title*” and write

$$\text{ISBN} \rightarrow \text{title}$$

This is called a *functional dependency* and analysis of functional dependencies (the subject of the video BCNF) is how databases are normalized and redundancy eliminated.

## Question 4

---

Write a single SQL query that lists, for each combination of library and author, the *name* of the library, the *author*, and the *total number of books* by that author currently on loan from that library.

```
SELECT name, author, SUM(onLoan)
FROM stocks JOIN BookEdition USING (ISBN)
GROUP BY name, author;
```

## Question 5

---

Write a single SQL query that finds the cheapest seller (or sellers) for every book edition. Each row should contain the ISBN, the book title, the cheapest available price for that edition, and the URL of the seller. If there are two equal-cheapest sellers, then there should be one output row for each of them.

```
SELECT ISBN, title, price, url
FROM BookEdition
JOIN sells USING (ISBN)
JOIN Seller USING (name)
WHERE (ISBN, price) IN
  (SELECT ISBN, MIN(price)
   FROM sells
   GROUP BY ISBN);
```

## Question 6

---

Write a single SQL query that lists the ISBN, title and author for each book that is not stocked by any libraries in the database.

```
SELECT ISBN, title, author  
FROM BookEdition  
WHERE ISBN NOT IN  
(SELECT ISBN FROM stocks);
```

# Recent lectures

---

- ▶ Week 7
  - ▶ Relational Algebra 2
  - ▶ Functional Dependencies
  - ▶ NULL
- ▶ Week 8
  - ▶ Data Integrity
  - ▶ ERD 3
  - ▶ DML (Data Manipulation Language)
- ▶ Week 9
  - ▶ Views
  - ▶ Views II
  - ▶ Triggers

## The word `NULL`

---

The word `NULL` is used to indicate *unknown* or *inapplicable*.

- ▶ An *arithmetic expression* involving a value that is `NULL` will evaluate to `NULL`, and
- ▶ A *boolean expression* involving a value that is `NULL` will evaluate to `NULL`, but
- ▶ Any `COUNT`, `SUM`, `AVG` function used on a list containing a `NULL` will *omit that row* from its calculations.

Note: `sqlite3` uses a blank space to represent `NULL` unless you use the dot command `.nullvalue <string>` to change it to `<string>`.

## Uses of NULL

---

The `LEFT JOIN`, or `LEFT OUTER JOIN` joins tables in such a way that *every row of the left table* appears in the result, even if it there is *no matching row* in the right table.

```
SELECT *
FROM Customers LEFT JOIN Orders
    USING (customerNumber);
```

One row of the output is

customerNumber	customerName	orderNumber	orderDate
125	Havel & Zbyszek Co	NULL	NULL

## Counting, including zeros

---

```
SELECT customerNumber, COUNT(orderNumber)
FROM Customers LEFT JOIN Orders
    USING (customerNumber);
```

- ▶ For any customer that *has* made orders, this counts them.
- ▶ For any customer that *has not* made orders,  
`COUNT(orderNumber)` is zero, because `NULL` does not contribute to the count.

This is the standard way to perform counts where you *do want* zeros to be included.

Note that `COUNT(*)` will count 1, rather than 0.

## Dealing with `NULL`

---

Often it is useful to check if a value is `NULL`, or not.

The normal comparison operators `==` and `!=` cannot be used, because both `mark == NULL` and `mark != NULL` evaluate to `NULL`.

There are two special comparison operators `IS NULL` and `IS NOT NULL` for this purpose.

```
SELECT COUNT(*)  
FROM people  
WHERE died IS NULL;
```

## Sets containing NULL

---

Be careful when using `IN` / `NOT IN` with column subqueries that may return a list containing a `NULL` value.

```
2 IN (1, 2, NULL)  
3 IN (1, 2, NULL)
```

The first of these is `1` (true), while the second is `NULL`.

```
2 NOT IN (1, 2, NULL)  
3 NOT IN (1, 2, NULL)
```

The first of these is `0` (false), while the second is `NULL`.

# Referential integrity

---

*“Teach them about referential integrity!”* (Ian C)

Relational databases store information about one entity set in *different tables*, each storing data about only one thing.

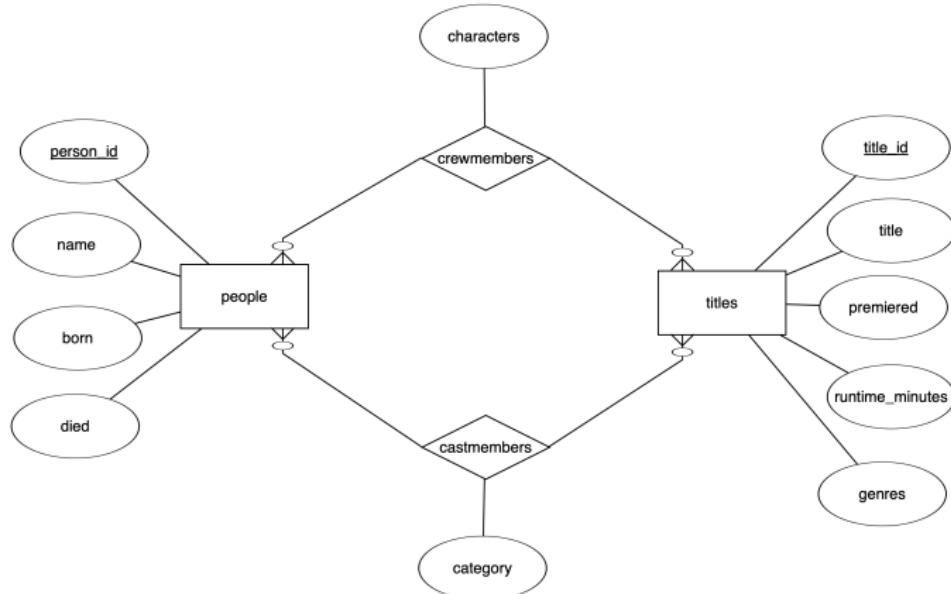
Queries usually have to *join* tables in order to answer all but the simplest questions.

Every row contains data relating to a particular entity, so there must be some way of indicating *which entity* — a value uniquely identifying the entity that is called a *reference* to the entity.

*Referential integrity* refers to the process of ensuring that these references are *internally consistent*.

# IMDB

---



## Valid references

---

```
CREATE TABLE crewmembers (
    title_id TEXT,
    person_id TEXT,
    category TEXT);
```

The *intention* is that title\_id should refer to a unique row of **titles**, and person\_id to a unique row of **people**.

This can be enforced through a *foreign key constraint* that “connects” the column in the child table with the corresponding column in the parent table.

## Foreign keys

---

A *foreign key constraint* tells SQLite about this connection.

```
CREATE TABLE crewmembers (
    title_id TEXT,
    person_id TEXT,
    category TEXT,
    FOREIGN KEY (title_id) REFERENCES titles(title_id),
    FOREIGN KEY (person_id) REFERENCES people(person_id) );
```

The system will henceforth ensure that title\_id and person\_id always refer to *legitimate rows* of **titles** and **people**.

## What effect does this have?

---

This has the effect of *forbidding* any database operation that would leave `title_id` or `person_id` referring to a *non-existent tuple* in either `titles` or `people`.

This includes

- ▶ Any `INSERT` or `UPDATE` statement on `crewmembers`, and
- ▶ Any `UPDATE` or `DELETE` statement on `titles` or `people`.

# Cascading

---

Referential integrity refers to *internal consistency* of a database.

So it can be legitimate to *alter* the values of, say title\_id, as long as *every reference* in *every table* to that title is also changed to reflect the new value.

There are four main options:

```
<foreign_key declaration> ON UPDATE CASCADE  
<foreign_key declaration> ON UPDATE SET NULL  
<foreign_key declaration> ON DELETE CASCADE  
<foreign_key declaration> ON DELETE SET NULL
```

## Warning

---

In sqlite3, you need to “turn on” enforcement of foreign keys

```
PRAGMA foreign_keys = ON;
```

In (my version) of SQLiteStudio, foreign keys are turned on already.

CITS1402  
Relational Database Management Systems

Seminar 9

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



# Test 1

---

Suppose that  $R(A, B)$  and  $S(B, C)$  are two tables and that currently they have the following contents

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

How many rows are in the table produced by the query

```
SELECT *
FROM R NATURAL JOIN S;
```

(Average score: 2.7 / 5.0)

## Test 2

---

Consider a university database storing details about venues (i.e., lecture theatres, labs and classrooms) and their capacity (number of students that can fit into the venue): these details are stored in a table `Venue(vName, capacity)`.

Which of the following three SQLite queries will produce the name and capacity of the venue (or venues, if there is more than one) that have the greatest capacity?

```
# Query 1
SELECT vName, MAX(capacity)
FROM Venue;

# Query 2
SELECT vName, capacity
FROM Venue
WHERE capacity = (SELECT MAX(capacity)
                   FROM Venue);

# Query 3
SELECT V.vName, V.capacity
FROM Venue V
WHERE NOT EXISTS (SELECT *
                   FROM Venue V2
                   WHERE V2.capacity > V.capacity);
```

Average score: 1.7 / 5.0

Most common incorrect answer: Q1 and Q2 only.

# Test 3

---

Consider two tables **Tutor** and **Hours** with the following contents

id name		id	unit	hours
1	Sam	1	CITS1401	5
2	Tan	3	CITS1401	8
3	Sam	2	CITS1402	7
4	Deng	2	CITS1402	2
		3	CITS1402	6
		1	MATH1001	5
		3	MATH1001	7

How many rows are returned by the following query?

```
SELECT Tutor.id, SUM(hours)
FROM Tutor, Hours
WHERE Tutor.id = Hours.id
GROUP BY name;
```

Average score: 1.0 / 5.0

# Test 4

---

- . Consider a table with schema

```
Grade(studentId INTEGER, unit TEXT, mark INTEGER)
```

which contains data about the grades obtained for various units by a student. Students are uniquely identified by the `studentId` and the table contains no `NULL` entries.

What information is returned by the following query?

```
SELECT studentId, COUNT(*), MAX(mark)
FROM Grade
GROUP BY studentID
HAVING COUNT(*) > 1;
```

- (a) For each student in the table, it lists their ID, the number of units they have taken and the highest grade they have obtained.
- (b) For each student who has taken more than one unit, it lists their ID, the number of units they have taken and the highest grade they have obtained.
- (c) For each student that has taken more than one unit, it lists the student ID, the name of their best unit and their mark for that unit.
- (d) For each unit in the table with more than one student, it lists the number of students who took that unit, and the highest mark any student obtained in that unit.
- (e) For each unit in the table, it lists the number of students who took that unit, and the highest mark any student obtained in that unit.

Average score: 4.6 / 5.0

## Test 5

---

A lecturer is trying to enter marks into a spreadsheet with columns:

**Number, Name, Address, Unit, Mark**

She is frustrated because she doesn't know her students' addresses, and has to just leave that field blank.

What database management issue is she experiencing?

- Redundancy
- Deletion Anomaly
- Insertion Anomaly
- Update Anomaly
- CRUD

Average score: 4.0 / 5.0

# Test 6

---

Consider a database **LiveableCities** that stores the historical details of the 10 “most-liveable cities” according to the annual study published by the *The Economist* magazine.

yr	city	country
2022	Vienna	Austria
2022	Copenhagen	Denmark
..		
2022	Melbourne	Australia
2021	Auckland	New Zealand
2021	Osaka	Japan
..		

The database covers a number of years, and for each year, there are 10 rows corresponding to the 10 most liveable cities for that year.

Which of the following queries lists the *years* (once each), in which no city from Australia or Canada makes the top 10.

```
# Query 1
SELECT DISTINCT yr FROM LiveableCities
WHERE yr NOT IN
(SELECT yr FROM LiveableCities
 WHERE country IN ('Australia', 'Canada'));
```

```
# Query 2
SELECT yr FROM LiveableCities
EXCEPT
SELECT yr FROM LiveableCities
 WHERE country in ('Australia', 'Canada');
```

```
# Query 3
SELECT DISTINCT yr FROM LiveableCities
 WHERE country NOT IN ('Australia', 'Canada');
```

## Test 6 cont

---

Breakdown was

Q1 and Q2 only – 20%

Q1 and Q3 only – 47%

Q2 and Q3 only – 8%

All three – 28%

None – 6%

This was the hardest question.

## Test 7

---

Suppose that  $R(A, B, C)$  and  $S(C, D, E)$  are relations containing the following tuples:

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

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

How many tuples will be in the following relation?

$$\pi_{A,B}(R) \bowtie_{R.A=S.E} \pi_{D,E}(S)$$

Average: 1.2 / 5.0

## Test 7

---

The output is a join of two tables, so

1. Calculate  $\pi_{A,B}(R)$  by *projection*
2. Calculate  $\pi_{D,E}(S)$  by *projection*
3. *Join* the tables using the condition  $R.A = S.E$

Unpack the expression, remember that the *only* relational operators are  $\pi$ ,  $\sigma$ ,  $\bowtie$  and  $\bowtie_c$ , eliminate duplicate rows and proceed systematically.

# Test 8

---

Consider the `Enrolment` table from the university database (as in lectures) which has the following schema:

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

What does the following query produce? (Here 'CITS1402' is the unit code for Databases.)

```
SELECT sid  
  FROM Enrolment  
 WHERE mark >= 50  
   AND uCode == 'CITS1402';
```

Average of 4.73 / 5 (was the easiest question).

# A real(ish) data set

---

Dataset about NY City Parking Tickets

```
CREATE TABLE ParkingTicket (
    id INTEGER PRIMARY KEY,
    rego TEXT,
    state TEXT,
    day TEXT,
    carBody TEXT,
    carMake TEXT,
    carColour TEXT,
    carYear INTEGER
);
```

(See “Extra Practice Lab” for database and extra questions).

## Examine database

---

## White and black cars

---

List states with more white-car parking tickets than black-car parking tickets.

This will list the states and number of white-car parking tickets

```
SELECT state, COUNT(*) as white_tickets
FROM ParkingTicket
WHERE carColour IN ('WH', 'WHITE', 'WHI', 'WHT')
GROUP BY state;
```

## White and black cars

---

This will list the states and number of white-car parking tickets and number of black-car parking tickets

```
SELECT * FROM
(SELECT state, COUNT(*) as white_tickets
FROM ParkingTicket
WHERE carColour IN ('WH', 'WHITE', 'WHI', 'WHT')
GROUP BY state)
JOIN
(SELECT state, COUNT(*) as black_tickets
FROM ParkingTicket
WHERE carColour IN ('BK', 'BLK', 'BLACK') GROUP BY state)
USING (state);
```

## Add the condition

---

```
SELECT state FROM
(SELECT state, COUNT(*) AS white_tickets
FROM ParkingTicket
WHERE carColour IN ('WH', 'WHITE', 'WHI', 'WHT')
GROUP BY state)
JOIN
(SELECT state, COUNT(*) AS black_tickets
FROM ParkingTicket
WHERE carColour IN ('BK', 'BLK', 'BLACK') GROUP BY state)
USING (state)
WHERE white_tickets < black_tickets;
```

## Separating rows

---

This is awkward because we have to *separate out* the white-car rows and the black-car rows in order to count them.

Let's change viewpoint a bit — replace *counting* a special *subset* of rows with *adding up* a special *value* from *every* row.

The value `IIF(carColour IN ('WH', 'WHITE', 'WHI', 'WHT'))` is 1 for white-car rows, and 0 for every other row.

## The counting version

---

```
SELECT state,
SUM(IIF(carColour IN ('WH','WHITE','WHI', 'WHT'),1,0)) AS
white_tickets,
SUM(IIF(carColour IN ('BK','BLK','BLACK'),1,0)) AS
black_tickets
FROM ParkingTicket GROUP BY state;
```

This only needs to process the large `ParkingTicket` table once.

## Zeros

---

Our first solution will only count states where there is at least one white-car ticket and one black-car ticket, whereas this method includes zeros.

```
145 SELECT state,
146 SUM(IIF(carColour IN ('WH', 'WHITE', 'WHI', 'WHT'),1,0)) AS white_tickets,
147 SUM(IIF(carColour IN ('BK', 'BLK', 'BLACK'),1,0)) AS black_tickets
148 FROM ParkingTicket GROUP BY state
149 HAVING white_tickets = 0 OR black_tickets = 0;
```

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

- Toolbar:** Includes icons for refresh, checkmark, delete, back, forward, and search, followed by a page number input (1) and a refresh icon.
- Status Bar:** Displays "Total rows loaded: 5".
- Grid View Buttons:** "Grid view" (selected) and "Form".
- Table Headers:** state, white\_tickets, black\_tickets.
- Data Rows:** 5 rows of data:

	state	white_tickets	black_tickets
1	AB	3	0
2	FO	0	1
3	MB	0	1
4	NB	0	0
5	NS	1	0

## Quick and dirty

---

Because SQLite just uses `1` for true, and `0` for false, we don't even need the `IIF`.

```
SELECT state,
       SUM(carColour IN ('WH', 'WHITE', 'WHI', 'WHT')) AS
           white_tickets,
       SUM(carColour IN ('BK', 'BLK', 'BLACK')) AS black_tickets
  FROM ParkingTicket GROUP BY state;
```

But code that relies on specific SQLite kludges is neither robust nor very readable.

CITS1402  
Relational Database Management Systems

Seminar 10

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## This week

---

- ▶ Project Overview
- ▶ Triggers
- ▶ **CHECK** Constraints

# Principles

---

- ▶ Project is *very short*
- ▶ Project involves *fundamental concepts*
- ▶ Project emphasizes *understanding* over *busywork*
- ▶ Project consists of 7 separate *discrete parts*

The ERD is hand-marked while the rest is analysed by scripts that examine both the structure and operation of the code.

## To get marks

---

For the files other than ERD.png

- ▶ Submission

Submit plain text files with the correct names by the due date

- ▶ Operation

Files should create a table, trigger or view that meets the specification when tested. In particular, if reading in a file causes a *syntax error*, then it cannot be tested.

- ▶ Code Analysis

Marks may be reduced for code that is blatantly inefficient or overcomplicated

We are asking for a **small amount** of high-quality work

# Project Overview I

---

- ▶ **BookEdition**

Information about a particular edition of a book which is uniquely identified by the ISBN. SCML may have zero, one or many *actual physical copies* of each book in this table.

- ▶ **BookCopy**

An actual *physical book* which is uniquely identified by an **ISBN** and **copyNumber**. Other than this uniqueness, there are no particular constraints on the **copyNumber**.

# Project Overview II

---

- ▶ **Client**

Information about the clients who reside in various aged-care residences. Identified by a client id. Clients are entered into **Client** before they borrow any books.

- ▶ **loan**

Information about loans of particular copies of books to clients. Each row lists the client id, the ISBN and copy number of the book and two dates.

# Project Overview

---

```
CREATE TABLE loan (clientId INTEGER,  
ISBN TEXT,  
copyNumber INTEGER,  
dateOut TEXT,  
dateBack TEXT)
```

## Book is loaned out

---

When a client borrows a book, the mobile librarian inserts a new row into `loan`

```
INSERT INTO TABLE loan  
VALUES(112, '9780593607695' ,5, '2022-10-02' ,NULL);
```

*Except as specified*, (i.e., the requirement that the client and book copy must actually exist) you may assume that the values all make sense.

## Book is returned

---

When a client *returns* a book, the clerk *updates* that row in `loan`.

```
UPDATE loan
SET dateBack = '2022-10-10'
WHERE ISBN = '9780593607695' AND copyNumber = 5 AND
      dateBack IS NULL;
```

(There was a typo in the draft project document which will be corrected.)

You may assume that the newly entered `dateBack` makes sense.

## The trigger

---

Your *trigger* should fire on the `UPDATE` event.

The trigger should then calculate the *number of days* between the two dates.

The trigger should then add this value to the `daysLoaned` column for right book copy in `BookCopy`.

# Triggers

---

Here is the trigger syntax:

```
CREATE TRIGGER <triggerName> <timeSpec> <triggerAction>
ON <tableName>
FOR EACH ROW
BEGIN
...
...
END
```

For our purposes, the name of the trigger is irrelevant.

## Actions

---

The possible actions specified when creating a trigger are `INSERT`, `UPDATE` or `DELETE`

A trigger *fires* when the *specified action* occurs on the *named table*.

In other words, an `UPDATE` trigger will be fired whenever an `UPDATE` statement is executed.

The statements between `BEGIN` and `END` are then executed.

## Times

---

The choices for *time* are BEFORE, AFTER or INSTEAD OF.

Mentally, we imagine a BEFORE UPDATE trigger being fired *immediately before* the update is actually performed.

The trigger *intercepts* the UPDATE statement in order to validate, verify, correct, complete, or log the update.

Similarly, we imagine an AFTER UPDATE trigger firing *immediately after* the update has been performed.

For somewhat technical reasons, SQLite recommends using AFTER UPDATE rather than BEFORE UPDATE.

## What can it do?

---

Once a trigger has been fired, the statements between `BEGIN` and `END` are executed.

An `AFTER UPDATE` trigger has access to

- ▶ The database, where one row has just been updated, and
- ▶ A special tuple called `OLD` which is a copy of the original (pre-update) row, and
- ▶ A special tuple called `NEW` which is a copy of the current (post-update) row.

In our situation `OLD.dateBack` would be `NULL` and `NEW.dateBack` would be '`2022-10-10`'.

## Check Constraints

---

A `CHECK` constraint is a *boolean expression* that is associated with either a column, or the whole table.

SQLite blocks *any operation* that would cause this boolean expression to become false.

```
CREATE TABLE Student (
    sName TEXT,
    sNumber TEXT CHECK(length(sNumber == 8)),
    sEmail TEXT);
```

## More than one check

---

Checks can refer to more than one column and do not have to be declared next to a column name.

```
CREATE TABLE Student (
    sName TEXT,
    sNumber TEXT,
    sEmail TEXT,
    CHECK (length(sNumber) == 8),
    CHECK (instr(sEmail,'@') > 0));
```

But **CHECK** constraint cannot contain a subquery, so the most it can do is verify that the data in a single row passes some sort of consistency check

## Two project gotchas

---

- ▶ *Submitting files containing syntax errors*

The testing of your project involves running your code (to set up the tables, or the trigger, or the views) and then observing how they work with sample data and queries.

If your files contain syntax errors, or create tables or triggers or views with the *wrong names*, then the testing will not work.

- ▶ *Not following the specifications*

The project is deliberately small and focussed, e.g., you need only implement referential integrity on one table.

If you add features that are *not in the specification*, then they will either leave your mark unchanged, or actually reduce it (if they create a subtle error).

CITS1402  
Relational Database Management Systems

Seminar 11

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## This week

---

- ▶ Project Submission

## Plagiarism reminder

---

This is an *individual project*.

You should not *look at* anyone else's code, or *show* anyone else your code.

You may *verbally* discuss your approach in general terms, but should not look at anything *written* (on paper or a screen).

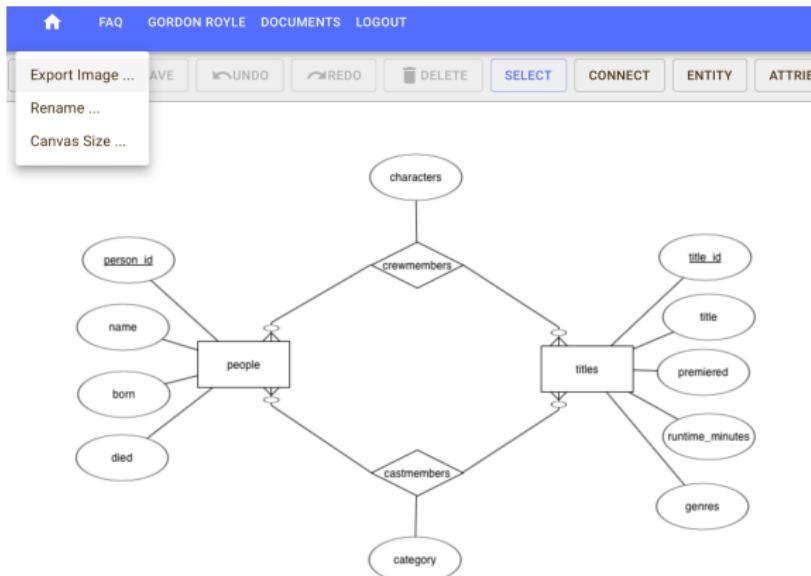
If *A* helps *B* inappropriately, then *A* and *B* are penalized *equally*.

*Students detected in plagiarism cases are often far more upset about costing their friend the 15% than about losing their own.*

# ERD.png

---

A file called ERD.png (watch out for hidden filename extensions)



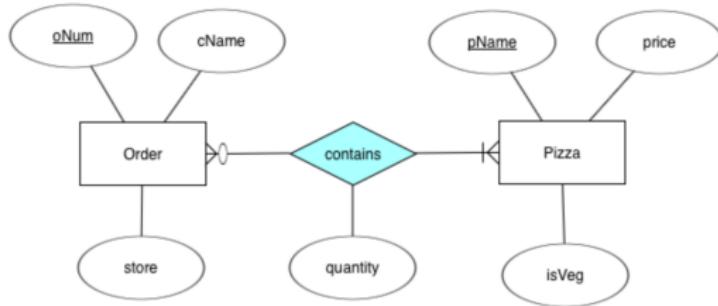
## Checklist

---

Watch out for:

- ▶ Every entity set becomes a table (of the same name)
- ▶ Every many-to-many relationship becomes a table
- ▶ A one-to-many relationship does not usually become a table
- ▶ Attributes on entity sets become columns of the table
- ▶ Relationship attributes on many-to-many relationships become columns of the table
- ▶ Primary keys indicated by underlining attribute(s)
- ▶ Foreign keys ***are not*** relationship attributes

# Pizza



Relationship Name

contains

Identifying

Entity One

Order

Mandatory

One

Optional

Many

Unspecified

Unspecified

Entity Two

Pizza

Mandatory

One

Optional

Many

Unspecified

Unspecified

Edit Exact Constraints

ADD ATTRIBUTE

## Four tables

---

The files BookEdition.sql, BookCopy.sql, loan.sql should each contain one **CREATE TABLE** statement (terminated by a semicolon)

```
CREATE TABLE Client (
    clientId INTEGER,
    name TEXT,
    residence TEXT,
    <your improvements>
);
```

You should just be *adding* your data integrity statements, and not changing anything else.

(They do not necessarily have to be located at exactly the same place as shown here.)

## Marking

---

Our marking program will use the most recent version of sqlite3, and will create the database something like this

```
PRAGMA foreign_keys = 1;  
.read 12345678/BookEdition.sql  
.read solutions/BookCopy.sql  
.read solutions/Client.sql  
.read solutions/loan.sql
```

This reads in *your file* for BookEdition, but our files for the rest.

## For example

---

One set of tests will involve the validation of the 5-digit ISBN invented for the project.

To get full marks, your code should

- ▶ *Accept* any tuple with a *valid* ISBN
- ▶ *Reject* any tuple with an *invalid* ISBN

A `CHECK` statement that works sometimes, but not always, will be given partial credit.

On the other hand, if your `BookEdition.sql` contains a *syntax error*, then we can't even test the parts that might work.

## Books Out

---

The testing script will test `loan` by simulating a small number (maybe 10 or 20) loans and returns.

Each new loan is entered by a statement of the form

```
INSERT INTO TABLE loan  
VALUES(11, '12344', 3, '2022-10-15', NULL);
```

You *do not* have to validate the `dateOut` or check that the `dateBack` is `NULL`.

## Rental Back

---

When a client *returns* a book, the mobile librarian *updates* that row in `loan`.

```
UPDATE loan
SET dateBack = '2022-10-29'
WHERE ISBN = '12344' AND copyNumber = 3 AND dateBack IS
NULL;
```

The date will be a correctly formed string representing a date after `dateOut` and so you *do not* have to validate or check it.

# Triggers

---

The skeleton outline of the trigger should be

```
CREATE TRIGGER <triggerName> AFTER UPDATE  
ON loan  
FOR EACH ROW  
WHEN ...  
BEGIN  
...  
...  
END
```

Your file should just *create* the trigger. As the trigger only fires when certain actions (updates) happen to the table, you will have to actually perform some **UPDATE** commands to test it.

## On firing

---

When a book is returned, the trigger fires *immediately after* the update, and has access to *two special tuples*

```
OLD = (11, '12344', 3, '2022-10-15', NULL)
NEW = (11, '12344', 3, '2022-10-15', '2022-10-29')
```

The trigger can *access* these values through expressions such as `OLD.ISBN` or `NEW.dateBack` etc.

The trigger should calculate the number of days in this just-completed loan, then *add* that value to the `daysLoaned` column of the *correct row* in `BookCopy`.

## Comments

---

When the trigger fires, it executes one (or more) SQL statements.

The statements must be *full SQL statements* that you could run yourself (if you filled in the values that come from `OLD` and `NEW`).

These statements can access *any tables* in the database in the same way as usual.

Video 25 DML covers data manipulation statements like `UPDATE`.

The trigger has *no concept* of a “current table” or “current row” so you cannot, for example, refer to `dateOut` or `dateBack`, and expect it to have any meaning.

## The view

---

The final part of the project is the file `readingHistory.sql` that should *create a view*, called `readingHistory`.

Recall that a *view* in SQL is a *virtual table* that is defined by a saved SQL statement.

When you actually run

```
.read readingHistory.sql
```

then this should not generate any output, but just silently create the view.

## Testing the view

---

To test that the view works, you need to run statements involving it, such as

```
SELECT * FROM ReadingHistory  
WHERE clientId = 11;
```

If the view is working, the system actually runs the saved statement to create `ReadingHistory` “*on-the-fly*”.

The remainder of the query runs as usual.

# Submission

---

The *soft deadline* is 1700 Sunday 16th October (next Sunday).

At this time, the “late-penalty clock” starts ticking (figuratively speaking).

Late penalties are assessed on a per-file basis using the *last submission time* for each file.

The *hard deadline* is 1700 Sunday 23rd October (end of Week 12).

No submissions can be accepted after that time.

## Extensions

---

In case of *sickness* or *misadventure*, you should apply for Special Consideration at your Student Office.

If your application is successful, then you may be granted *up to 7* days extension, in which case any late penalties will be adjusted accordingly.

UAAP students entitled to X days extension will have this applied automatically, but the hard deadline *still applies*.

CITS1402  
Relational Database Management Systems

Seminar 12

GORDON ROYLE

DEPARTMENT OF MATHEMATICS & STATISTICS



## This week

---

- ▶ Functional Dependencies
- ▶ Past Exam Paper

# Functional Dependencies

---

A *functional dependency* indicates relationships between columns arising from the underlying *business logic*.

StudentMarks(StudentNumber, Name, Email, Unit, Mark)

has the functional dependency  $\text{StudentNumber} \rightarrow \text{Name, Email}$

This means “if two rows have the same  $\text{StudentNumber}$  then they have the same  $\text{Name}$  and the same  $\text{Email}$ .

Terminology:  $\text{StudentNumber}$  *determines*  $\text{Name, Email}$ .

# Redundancy

---

Reasoning about FDs gives information about *possible redundancy*.

1001	Amy	amy@uwa.edu	CITS1401	55
1001	Amy	amy@uwa.edu	CITS1402	82
1002	Boo	boo@its.com	CITS1401	66
1002	Boo	boo@its.com	CITS1402	77
1003	Cai	cai@aaa.net	MATH1011	88
1003	Cai	cai@aaa.net	PHYS1001	44

Redundancy leads to various *anomalies* threatening data integrity.

## The destination

---

Ultimately we will decide to *decompose* this table into two tables

Student(StudentNumber, Name, Email)  
Marks(StudentNumber, Unit, Mark)

This decision is made through *reasoning about FDs*.

## Acceptable FDs

---

Some FDs are *inevitable* and do not cause any problems.

`Student(StudentNumber, Name, Email)`

still has the FD

$$\text{StudentNumber} \rightarrow \text{Name, Email}$$

This is OK because `StudentNumber` is a *key* for the table, so no two rows can have the same value.

Hence the anomalies just don't arise.

## Reasoning about FDs

---

Reasoning about FDs is normally done with *synthetic* examples.

Suppose  $R = (A, B, C, D, E, F)$  is a relation with FDs

$$C \rightarrow F, E \rightarrow A, CE \rightarrow D, A \rightarrow B$$

Which is the following is a *key* for  $R$ ?

$$CD, CE, AE, AC$$

## Consider $CE$

---

$C \rightarrow F, E \rightarrow A, CE \rightarrow D, A \rightarrow B$

$CE \rightarrow CE$  (trivial)

$CE \rightarrow ACE$  ( $E \rightarrow A$ )

$CE \rightarrow ACDE$  ( $CE \rightarrow D$ )

$CE \rightarrow ABCDE$  ( $A \rightarrow B$ )

$CE \rightarrow ABCDEF$  ( $C \rightarrow F$ )

# BCNF

---

A relation is in *Boyce-Codd Normal Form* if the only non-trivial functional dependencies arise from a *key*.

Once in BCNF, all redundancy based on functional dependencies has been removed.

A functional dependency that *violates* BCNF gives information about how to decompose the relation.

# Decomposition

---

StudentMarks(StudentNumber, Name, Email, Unit, Mark)

**StudentNumber → Name, Email**

Create two new relations, one containing the information from this FD, and one containing everything else.

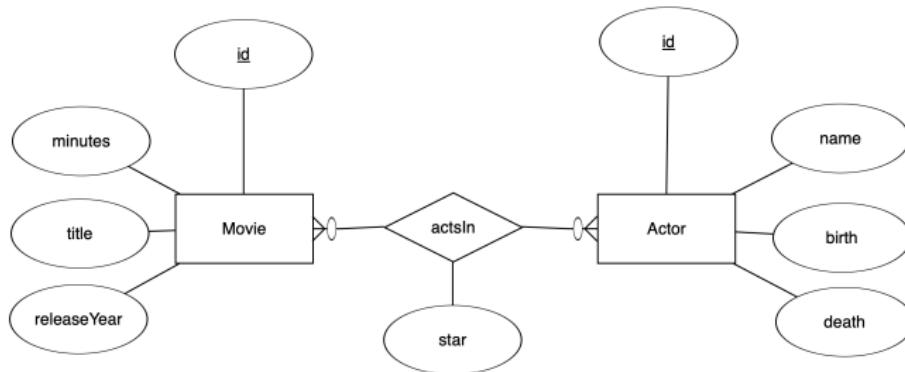
Student(StudentNumber, Name, Email)

Marks(StudentNumber, Unit, Mark)

Now, the first relation is in BCNF, but what about the second?

# 2019 Exam ERD

---



The table **actsIn** has schema

**actsIn(actorId, movieId, star)**

# Final Exam 1

---

List the titles of the movies released in 2019 that are more than 3 hours long.

```
SELECT title  
FROM Movie  
WHERE releaseYear = 2019  
AND minutes > 180;
```

(Testing basic single-table query with a simple boolean condition.)

## Final Exam 2

---

List the id of each actor, along with the *number* of movies in which that actor has appeared. (Do not include actors who have not appeared in any movies.)

```
SELECT actorId, COUNT(*)  
FROM actsIn  
GROUP BY actorId;
```

- ▶ Testing GROUP BY and COUNT(\*)
- ▶ Asking for `id` and not `name` and excluding zeros mean “Don’t need to use Actor”

## Final Exam 3

---

For each year represented in the database, list the year, the *number* of movies released in that year, and the *average length* of the movies released in that year.

```
SELECT releaseYear, COUNT(*), AVG(minutes)
FROM Movie
GROUP BY releaseYear;
```

(Testing summary function use)

## Final Exam 4

---

For each year represented in the database, list the year and the title of the *longest movie* released in that year

```
SELECT M.releaseYear, M.title
FROM Movie
WHERE M.minutes = (SELECT MAX(minutes)
                    FROM Movie
                    WHERE releaseYear = M.releaseYear);
```

```
SELECT releaseYear, title
FROM Movie
WHERE (releaseYear, minutes) IN
      (SELECT releaseYear, MAX(minutes)
       FROM Movie
       GROUP BY releaseYear);
```

Testing subqueries (either correlated or uncorrelated).

## Final Exam 5

---

List the *id* and *name* of all of the actors who have been a star in at least five movies.

```
SELECT id, name
FROM Actor JOIN actsIn ON (id = actorId)
WHERE star = 'Y'
GROUP BY id
HAVING COUNT(*) >= 5;
```

Notes: *name* is a bare column, but it is functionally dependent on *id* so this is not a problem.

## Final Exam 6

---

List the `id` and `name` of all of the actors who have *never been* a star in a movie.

```
SELECT id, name
FROM Actor
WHERE id NOT IN
    (SELECT actorId FROM actsIn WHERE star = 'Y');
```

- ▶ Testing `NOT IN` (or similar constructs).
- ▶ This includes actors not in any movies

## Final Exam 7

---

List the *id* and *name* of each actor along with the *number of movies* in which they have appeared. (This time you *do need* to consider actors who have not yet appeared in any movies.)

```
SELECT id, name, COUNT(movieId)
FROM Actor LEFT JOIN actsIn ON (id = actorId)
GROUP BY id;
```

- ▶ Must use `LEFT JOIN` to include actors not in any movies,
- ▶ Must use `COUNT(movieID)` (rather than `COUNT(*)`) to get zeros.

## Final Exam 8

---

List each movie id, along with the number of actors in that movie who are still alive and the number who are dead. (The output table should have three columns named `movieId`, `live` and `dead` with each row containing the relevant information for one movie.)

```
SELECT movieId,
       SUM(IIF (death IS NULL, 1, 0)) AS live,
       SUM(IIF (death IS NULL, 0, 1)) AS dead
  FROM actsIn JOIN Actor ON (id = actorId)
 GROUP BY movieId;
```

- ▶ Need `Actor` (for dead/alive) but not `Movie`.
- ▶ Using `IS NULL` for live/dead.
- ▶ Using “indicator function” method with `SUM` and `IIF` to do two different counts in one pass through the table.

## Back in 2019

---

In 2019, there were then three “short essay” type questions covering *transactions*, *data integrity* and *functional dependencies*.

Because they required quite a lot of writing, and I did not want to do the “scan-and-upload” style of exam, testing these topics was moved to the other parts of the exam.