

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

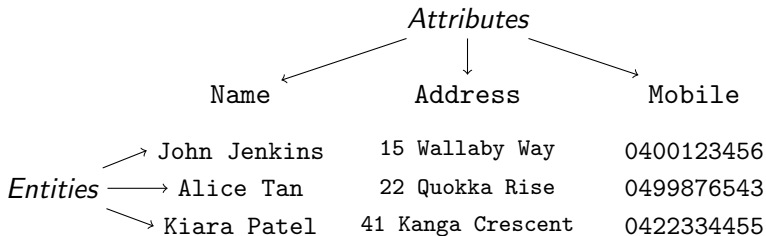
- ▶ The need to store data
- ▶ The basic CRUD operations
Create, *Read*, *Update* and *Delete*
- ▶ Dominance of the relational model

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.

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 as TextEdit 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 (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';
```

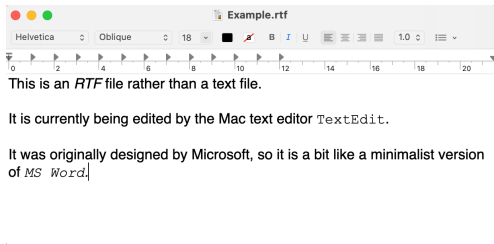
But the syntax of `OR` is

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

What is 'Fremantle' as a boolean expression?

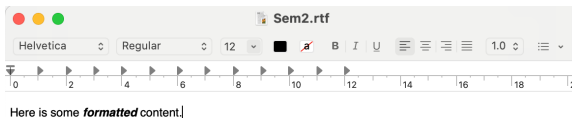
Gotcha 5 — Avoid RTF files

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\fs
iss\fcharset0 Helvetica-BoldOblique;}}
{\colortbl;\red255\green255\blue255;}
{\*\expandedcolortbl;;}
\paperw11900\paperh16840\margr1440\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 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

     1    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:

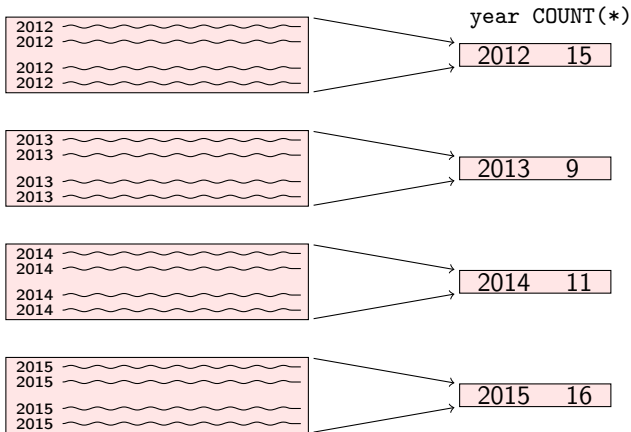
```
2012|1|Western Bulldogs|West Coast|87|136
2012|2|West Coast|Melbourne|166|58
...
2012|21|Port Adelaide|West Coast|50|98
2012|22|West Coast|Collingwood|107|58
```

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.

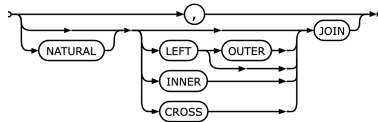
However, like most other implementations of SQL, SQLite assumes that the user *knows what they are doing* and uses the value of the bare column from the first row of the group.

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.

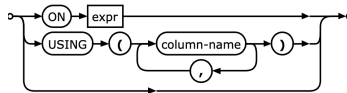


Ways to JOIN

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

$\text{people} \times \text{castmembers}$

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 Coe	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 Coe	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 Coe	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*.

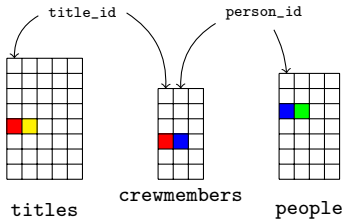
When devising a query, *be systematic*, and start by asking

- ▶ 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.

```
1 SELECT title
2 FROM titles JOIN ratings
3 WHERE votes > 2000000;
4 |
```

Grid view

Total rows loaded: 1500

	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

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* π
- ▶ The *selection operator* σ
- ▶ 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* π 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* σ 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

$$\pi_{A,B}(R) \times \sigma_{A=B}(R)?$$

A	B		A	B	C		A	B	C
1	1	\times	1	1	3	$=$	1	1	3
2	1		1	1	2		1	1	2
							2	1	2

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

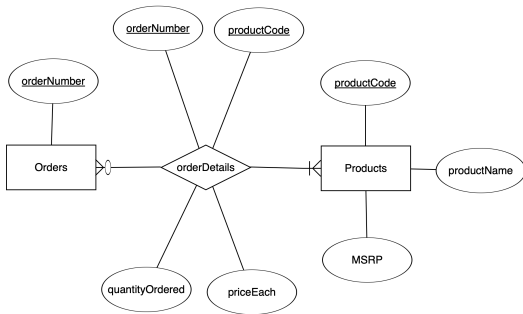
The database `classicmodels.db` is an SQLite database derived from an old widely-used teaching database created for MySQL.

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.



Diagrammed by ERDPlus.com

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




QueryHistory

1 SELECT * FROM orderDetails WHERE orderNumber = 10200;
2 |

Grid viewForm view



1



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

Corinthian Hills Temple drive
NAME : _____
ADDRESS : _____
TIN : _____
BUSINESS STYLE : _____
TRANSACTION CODE 1

1 HEALTHPLUSCANOLA0IL1	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	107.00V
1 J J PIATTOS CHEESE	27.00 V
2 J J V CUT BBQ 60 @24.25	48.50V
1 CLOVER CHEESE FAMILY	27.00 V
1 CEBUBRANDDRDMANGOSLI	115.00 V
0.088 V KINCHAY KG @392.00	34.50N
1.042 MIKISA CRAB CLAW @828.00	862.78N
0.504 MIKISA LAPU LAPU @738.00	371.95N
1 BOUNTY FRESH MEDIUM	92.50 N
1 SURFPOWDERPURPLEBLOO	162.50 V
6 SURFPOWDERPURPLE @5.00	30.00V
5 SBINTERLEAVEDBTR @54.10	270.50V
1 SBBATHROOMTISSEU1000	119.90 V
TOTAL	2,640.58
1 PESO	2,640.58
PROD CNT: 18 TOT QTY: 27.63	
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
1	1
2	2
2	0

B	C
1	3
2	2
1	2

How many rows are in the table produced by the query

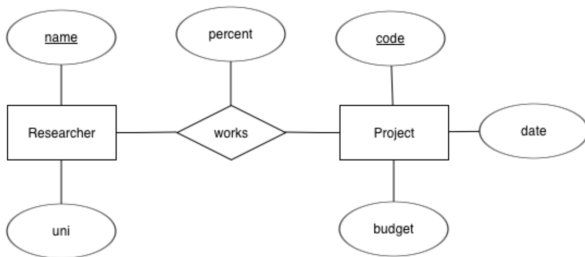
```
SELECT *  
FROM R JOIN S USING (B);
```

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?

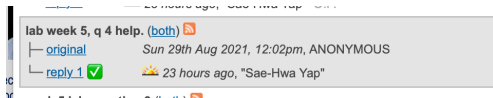
1. *Each* entity set becomes a table of the same name, whose columns are the attributes.
2. Only *some* relationships become tables (the many-to-many ones) while the others can be implemented via attributes.

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.)