Databases

If we write a program & execute it, all generated data will disappear once the program terminates, unless we specifically store it somewhere

SQL Databases are a sensible choice for where to save your data. They are:
- Highly optimised for the storage of tabular data
- Fast & well understood query language
- Fault tolerant protocols.

What is a Database?

Basically a super fancy spread sheet. Each database will contain tables that store data. The data in these tables can be queried using a language called SQL. Data from different tables can be joined to answer questions

Types of Database

Traditionally, the database would reside on a seperate machine & it would be managed by a database engineer. This is when data storage was expensive & if you wanted to access the DB you had to physically connect to it.

These days storage space is very cheap Meaning local, per-application databases are very common.

If you need remote data access you should use a server-style database like MariaDB, otherwise use a file-style database like Sqlite.

Server-style DB: Uses a client-server architecture. This means there is a dedicated database server that stores & manages the data & multiple client applications can interact with the database server to access & modify the data.

File-style DB: Data is stored & managed directly within the files on the file system of the OS, rather than being managed by a dedicated DB server.

Using SQLite

The SQLite package is called `sqlite3`
- make sure it is installed (`sudo apt install sqlite3`)
To initialise a DB simply do

```
sqlite3 NAME.db
```

You'll see this alters your command line to `sqlite>`. To leave the SQL environment just do:

```
.exit
```

Using (MySQL)/MariaDB

Both very similar, made by the same person. Command to access both is `mysql`

if you run `mysql` as a lone command it'll return an error saying it can't connect to local MySQL server.

This is because we need to start the server running first or say where to connect to. On Linux this needs to be done via SystemD (which is a method of managing system services - processes on Linux). Fortunately, this can simply be done via command line:

```
(sudo) systemctl start mariadb
(sudo) systemctl enable mariadb
```

Now we can run `mysql`.

Once Maria is up & running it'll have some test DB's & a root user with no passwords. It is up to us to secure it. `mysql_secure_installation` can automate most of this.

If you were being paid, some methods you could look at:

- Set usernames and passwords
- Firewall off ports
- Add logging and intrusion detection
- Backup
- Secure backups

Relational Modelling

Lecture 2/6

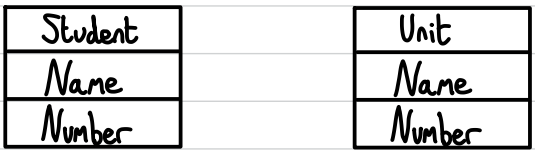
We saw that DBs store data in tables, now we're going to look at how to structure the data in a table so we can visualise it. This is Relational Modelling.

This is just a tool for thinking about how to decompose relationships between things in tables. So long as the meaning is clear it doesn't matter how we present the diagrams.

We'll see some similarities between OOP & this.

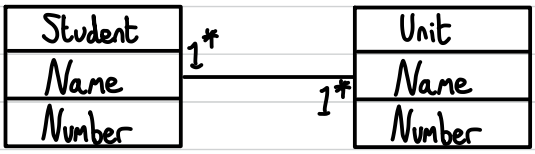
Things are nouns. Let's consider a student. A student has a name & a number. In this case, the student is the entity & the name & number are attributes

Now let's consider an academic unit. This will also have a Name & a Number:

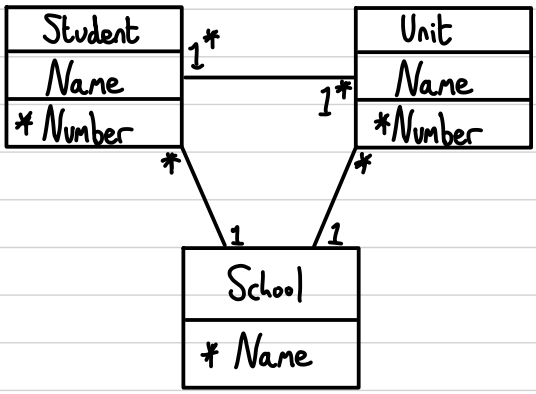


These classifications could take on any specific values, but we don't care about that now. We're looking at the relationships between entities.

This is where we see the Multiplicity from SE. One student can take (be associated with) many units & a unit can be associated with many students:



We can also have Schools (they're things too). Each student & unit belongs to a single school but a school has many students & units. The school will only have a name attribute.



On Entity Relationship diagrams, the primary key is denoted by an asterix/wildcard next to the relevant attribute

How do we determine unique entities?

We use a **key**. A key for an entity is the set of attributes needed to uniquely refer to it.

A **candidate key** is a minimal set of attributes needed to refer to it.
(Set of one or more attributes within a table that uniquely identifies each row in the table.)

The **primary key** for an entity is the key we use.
(The candidate key chosen by the designer to uniquely identify each row in the table)

(A primary key is selected from the candidate keys to serve as the main identifier for the table)

If a key contains multiple attributes it is called a **composite key**.

If a key is a meaningless ID column you added for the sake of having a key (UID) it is called a **surrogate key**

What Tables do we need to create?

Every entity becomes a table ~ each table has a primary key

Every edge on the Entity Relationship diagram becomes a table. The contents of these tables are the two primary keys being linked.
∧
An attribute that refers to another key is called a **foreign key**.

Intro to SQL

Lecture 3/6

SQL = Structured Query Language.

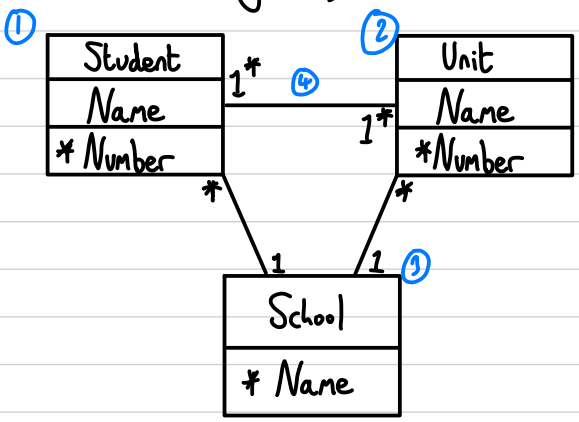
We are now going to look at how to create & query tables in databases.

- SQL is the dominant language for data queries. It is not a general purpose programming language. It is not Turing complete (cannot simulate a Turing machine / express any algorithm that can be described as a series of instructions)
- It has a very weird English-like syntax.

For the most part it is standardised but different database engines have slightly vary. SQLite is good with strings, most others prefer numbers. This Lecture focuses on SQLite Syntax.

① CREATE TABLE.

Let's use the system from the previous the lecture & the code to make the tables.




```sql
①CREATE TABLE IF NOT EXISTS student (
    name TEXT NOT NULL,
    number TEXT NOT NULL,
    PRIMARY KEY (number));

②CREATE TABLE IF NOT EXISTS unit (
    name TEXT NOT NULL,
    number TEXT NOT NULL,
    PRIMARY KEY (number));

③CREATE TABLE (IF NOT EXISTS)* school (
    name TEXT NOT NULL,
    PRIMARY KEY (name));

④CREATE TABLE IF NOT EXISTS class_register (
    student TEXT NOT NULL,
    unit TEXT NOT NULL,
    FOREIGN KEY (student) REFERENCES student(number),
    FOREIGN KEY (unit) REFERENCES unit(name),
    PRIMARY KEY (student, unit));
```

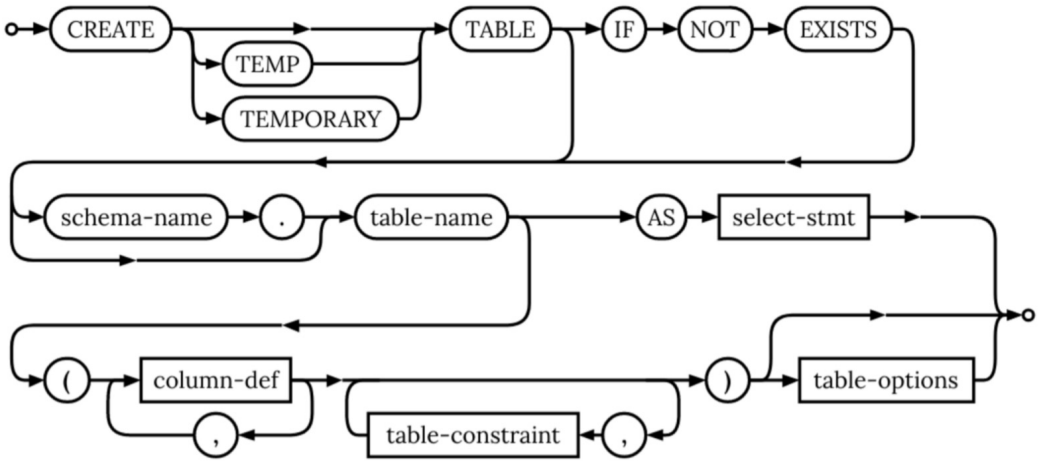
* While these conditionals are technically optional, it is recommended you keep them - if you try & delete a non-existant table it will throw an error.

② DROP TABLE

This is how we delete tables

```sql
DROP TABLE (IF EXISTS)* class_register;
DROP TABLE IF EXISTS student;
DROP TABLE IF EXISTS unit;
DROP TABLE IF EXISTS school;
```

(1.5) CREATE TABLE Syntax.



This tree diagram shows the syntax options for the CREATE TABLE function.

(3) Types

So far we've only seen TEXT but there are more:

INTEGER	whole numbers	
REAL	lossy decimals	
BLOB	binary data (images/audio/files...)	
VARCHAR(10)	a string of 10 characters	(TEXT)
TEXT	any old text	
BOOLEAN	True or false	(INTEGER)
DATE	Today	(TEXT)
DATETIME	Today at 2pm	(TEXT)

(4) Table constraints

Earlier we marked columns as NOT NULL (meaning those columns must take a non-null value)

We also saw PRIMARY KEY which must be unique + not NULL + potentially auto-generated

also FOREIGN KEY which ensures other key is not NULL

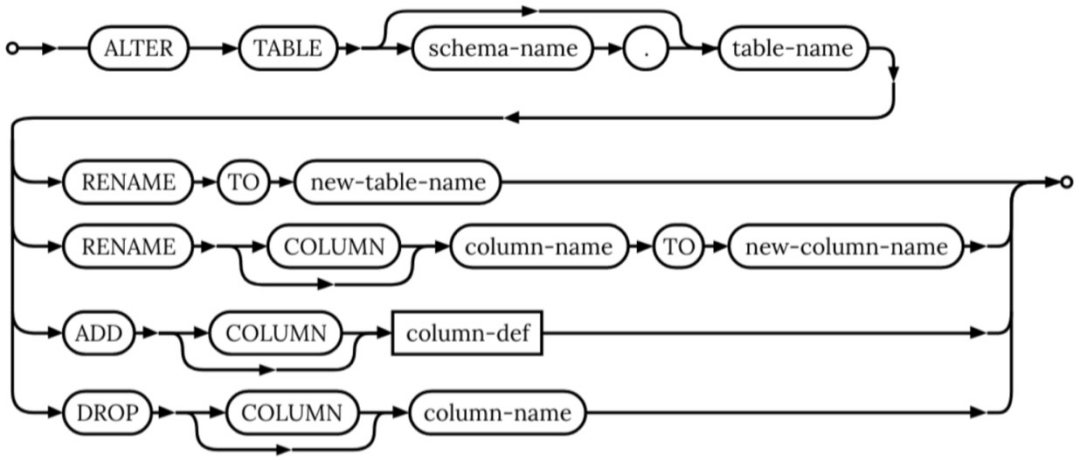
SQLite won't enforce constraints unless you tell it to using the STRICT keyword.

Constraints list:

NOT NULL	can't be NULL
UNIQUE	can't be the same as another row
CHECK	arbitrary checking (including it conforms to a regular expression)
PRIMARY KEY	unique, not NULL and (potentially) autogenerated
FOREIGN KEY	(IGNORED BY MARIADB) other key must exist

(5) ALTER TABLE

Even though it is often easier to save the table somewhere else, drop it & reimport it, here is the syntax tree:



(6) INSERT INTO

This is how you actually add data into a table.

```
INSERT INTO unit(name, number)
VALUES ("Software Tools", "COMS100012");
```

First line defines what column(s) you're filling

Second defines the values for those fields.

So far we've looked at how to create, delete & insert data into tables. Now let's look at how to query data

(7) SELECT

This is the basic command for selecting rows from a table.

```
SELECT * FROM album
LIMIT 5;
```

AlbumId	Title	ArtistId
1	For Those About To Rock We Salute You	1
2	Balls to the Wall	2
3	Restless and Wild	2
4	Let There Be Rock	1
5	Big Ones	3

```
SELECT * FROM artist
LIMIT 5;
```

ArtistId	Name
1	AC/DC
2	Accept
3	Aerosmith
4	Alanis Morissette
5	Alice In Chains

⑧ JOIN

This combines 2 tables:

The ON keyword specifies the condition that determines how the tables are related.

It typically involves specifying columns from each table that should match for the rows to be included in the result set.

```
SELECT *
FROM album
JOIN artist
ON album.artistid = artist.artistid
LIMIT 5;
```

AlbumId	Title	ArtistId	ArtistId	Name
1	For Those About To Rock We Salute You	1	1	AC/DC
2	Balls to the Wall	2	2	Accept
3	Restless and Wild	2	2	Accept
4	Let There Be Rock	1	1	AC/DC
5	Big Ones	3	3	Aerosmith

⑨ Reducing the columns.

How do we get rid of columns we don't want?

Only select the ones we need

```
SELECT album.title, artist.name
FROM album
JOIN artist
ON album.artistid = artist.artistid
LIMIT 5;
```

Title	Name
For Those About To Rock We Salute You	AC/DC
Balls to the Wall	Accept
Restless and Wild	Accept
Let There Be Rock	AC/DC
Big Ones	Aerosmith

⑩ Rename columns in the result set.

Use the AS keyword to assign an alias to a column/table in a query. This can improve the readability of the results.

AS when used in SELECT creates column alias' (because you're selecting columns)...

You can also give table alias'

```
SELECT album.title AS album,
       artist.name AS artist
FROM album
JOIN artist
ON album.artistid = artist.artistid
LIMIT 5;
```

album	artist
For Those About To Rock We Salute You	AC/DC
Balls to the Wall	Accept
Restless and Wild	Accept
Let There Be Rock	AC/DC
Big Ones	Aerosmith

```
SELECT e.first_name, e.last_name
FROM employees AS e
JOIN departments AS d
ON e.department_id = d.department_id
```

...But if it is used in FROM or JOIN it creates a table alias as FROM & JOIN specify tables

That is why the alias defined in SELECT doesn't clash with the table names in FROM & JOIN.

11 Filtering

WHERE is a clause used to filter rows returned by SELECT, UPDATE or DELETE based on a certain condition.
Let's filter on a certain word using `LIKE`, Usage:

WHERE [column name] LIKE [pattern]

```
SELECT album.title AS album,
       artist.name AS artist
FROM album
JOIN artist
ON album.artistid = artist.artistid
WHERE album LIKE '%Rock%'
LIMIT 5;
```

album	artist
For Those About To Rock We Salute You	AC/DC
Let There Be Rock	AC/DC
Deep Purple In Rock	Deep Purple
Rock In Rio [CD1]	Iron Maiden
Rock In Rio [CD2]	Iron Maiden

% Matches any sequence of zero or more characters

_ (underscore) Matches any single character.

LIKE is case-sensitive.

in SQL you can do not equal (!=) using: <>

12 Getting Unique Results

Here we use DISTINCT

```
SELECT artist.name AS artist
FROM album
JOIN artist
ON album.artistid = artist.artistid
WHERE album.title LIKE '%Rock%'
LIMIT 5;
```

artist
AC/DC
AC/DC
Deep Purple
Iron Maiden
Iron Maiden

```
SELECT DISTINCT artist.name AS artist
FROM album
JOIN artist
ON album.artistid = artist.artistid
WHERE album.title LIKE '%Rock%'
LIMIT 5;
```

artist
AC/DC
Deep Purple
Iron Maiden
The Cult
The Rolling Stones

At this point I've noticed a good way to workout what's going on is to add brackets

SELECT plucks out certain columns from the result of a query it returns then.

13 Counting occurances

Here we use COUNT & GROUP BY

```
SELECT artist.name AS artist,
       COUNT(album.title) as albums
FROM album
JOIN artist
ON album.artistid = artist.artistid
WHERE album.title LIKE '%Rock%'
GROUP BY artist
LIMIT 5;
```

artist	albums
AC/DC	2
Deep Purple	1
Iron Maiden	2
The Cult	1
The Rolling Stones	1

GROUP BY is used to group rows that have the same values into summary rows, typically to perform aggregate functions such as

COUNT, MAX, MIN, SUM, AVG... on grouped data

When you use GROUP BY, 2 things happen:

1. Rows with the same value for the specified column are grouped together
2. The aggregate function specified in the SELECT clause is applied to each of these groups.

(14) Ordering number of occurances.

This simply involves adding an ORDER BY clause. This orders the rows according to the values of a specific column.

It sorts the result set of a query

```
SELECT artist.name AS artist,
       COUNT(album.title) as albums
FROM album
JOIN artist
ON album.artistid = artist.artistid
WHERE album.title LIKE '%Rock%'
GROUP BY artist
ORDER BY albums DESC (ASC for asced)
LIMIT 5;
```

artist	albums
Iron Maiden	2
AC/DC	2
The Rolling Stones	1
The Cult	1
Deep Purple	1

Database Normal Forms

Lecture 4/6

So far we've looked at the mechanisms for designing & building databases. Now we're going to step back & look at some database theory.

This theory gives rules for designing tables that ensure certain properties that make the database easier to query, faster & avoid redundant data so that when you have update values, you only have to do it once

These rules are called Normal Forms. GPT: They are a series of guidelines designed to ensure that database tables are structured in a way that minimizes redundancy & dependency, leading to better organisation, maintainability & integrity of the data

First Normal Form (1NF)

Each column shall contain one & only one value.

Artist	Albums
The Beatles	Yellow Submarine, White Album, Rubber Soul
Milk Can	Make It Sweet
Dresden Dolls	Yes Virginia, No Virginia, The Dresden Dolls

✗

Artist	Album
The Beatles	Yellow Submarine
The Beatles	White Album
The Beatles	Rubber Soul
Milk Can	Make It Sweet
Dresden Dolls	Yes Virginia
Dresden Dolls	No Virginia
Dresden Dolls	The Dresden Dolls

✓

Second Normal Form (2NF)

1NF + Every non-key attribute is fully dependent on the key

If this isn't the case, make another table!

Artist	Album	Year	Prime Minister
The Beatles	Yellow Submarine	1969	Harold Wilson
The Beatles	White Album	1968	Harold Wilson
The Beatles	Rubber Soul	1965	Harold Wilson
Milk Can	Make It Sweet	1999	Tony Blair
Dresden Dolls	Yes Virginia	2006	Tony Blair
Dresden Dolls	No Virginia	2008	Gordon Brown
Dresden Dolls	The Dresden Dolls	2003	Tony Blair

✗

(keys are Artist & Album)
PM has nothing to do with Artist or Album (isn't dependent)

① ✓

Artist	Album	Year
The Beatles	Yellow Submarine	1969
The Beatles	White Album	1968
The Beatles	Rubber Soul	1965
Milk Can	Make It Sweet	1999
Dresden Dolls	Yes Virginia	2006
Dresden Dolls	No Virginia	2008
Dresden Dolls	The Dresden Dolls	2003

② ✓

Year	Prime Minister
1969	Harold Wilson
1968	Harold Wilson
1965	Harold Wilson
1999	Tony Blair
2006	Tony Blair
2008	Gordon Brown
2003	Tony Blair

Third Normal Form (3NF)

2NF + Every non-key attribute must provide a fact about the key, the whole key & nothing but the key.

Let's look at the prime ministers again

Year	Prime Minister	Birthday
1969	Harold Wilson	1916-03-11
1968	Harold Wilson	1916-03-11
1965	Harold Wilson	1916-03-11
1999	Tony Blair	1953-05-06
2003	Tony Blair	1953-05-06
2006	Tony Blair	1953-05-06
2008	Gordon Brown	1951-02-20

X

Our key is (Year, Prime Minister). So every non-key depends on the key ∴ 2NF.

But the Birthday doesn't tell you something about the whole key. Just the PM. ∴ Not 3NF

So split!!

Year	Prime Minister
1969	Harold Wilson
1968	Harold Wilson
1965	Harold Wilson
1999	Tony Blair
2003	Tony Blair
2006	Tony Blair
2008	Gordon Brown

Prime Minister	Birthday
Harold Wilson	1916-03-11
Tony Blair	1953-05-06
Gordon Brown	1951-02-20

Now, if we need to alter a value in the Birthday column we only have to change it once whereas before we would've had to do it multiple times.

GPT: "A table is in 3NF if it is in 2NF & if all non-key attributes are fully functionally dependent on the primary key & no non-key column depends on another non-key column.

Boyce-Codd Normal Form (BCNF)

This is a slightly stronger form of 3NF, sometimes called 3.5th Normal Form.

Every possible candidate key for a table is also in 3NF.

GPT: In simpler terms, BCNF ensures that every non-trivial dependacy in a relation is directly related to the primary key. If any attribute is functionally dependent on another attribute that is not part of the primary key, it needs to be decomposed into multiple tables to be in BCNF.

In mathematical terms, for any functional dependacy $X \rightarrow Y$ (X derives Y) (Y is dependent on X) that exists within a table, X must be a super key (a set of attributes that uniquely identifies each tuple of of a relation)

More on BCNF:

For any dependancy $A \rightarrow B$, A should be a super-key

Which means for $A \rightarrow B$ if A is non-prime & B is a prime, the table is not in BCNF

Up until now we've seen

Prime attributes ⟶ non-prime attributes
Functional Dependancy

Part of Prime attribute ⟶ non-prime attribute
Partial Dependancy (violates 2NF)

& non-prime attribute ⟶ non-prime attribute
Transtive Dependancy (violates 3NF)

Now: non-prime attribute ⟶ prime attribute
This violates BCNF

Example:
Let's say we have a table with the headers,

Student_id Subject professor (where a subject has multiple professors

Primary key (student_id, subject) as this uniquely identifies all rows. BUT, you can also use the professor to determine the subject. This gives:

& (student_id, subject) ⟶ professor
professor ⟶ subject

There are no partial or transitive dependancies here ∴ it is in 3NF.

But here a non-prime can determine a prime key
professor is not a superkey as it cannot uniquely classify each row (where student_id & subject can) ∴ this is not in BCNF.

So it needs breaking.

4th Normal Form (4NF)

BCNF + If multiple attributes are dependent on the same key, those attributes should be dependent to, otherwise split them into different tables

It must not contain more than one Multivalued dependency (MVD)

Definition of MVD: If we have the MVD $X \twoheadrightarrow Y$, it means for any two tuples in the table that share the same value for the X attribute must also share values for the Y attribute regardless of any other attribute values.

5th Normal Form (5NF)

4NF + It cannot be further non-loss decomposed. (Can't be split into more tables)

I'm not even going to bother.

Intermediate SQL

Lecture 5/6

NULL & JOINs

NULL is literally nothing, it is a special value used in SQL to represent missing data. They are pretty much always a bad idea & makes some comparisons difficult.

Consider this simple table:

Person	Fruit
Joseph	Lime
Matt	Apple
Partha	

Partha has no value for her favourite fruit. How do we query this table?

```
SELECT * FROM fruit WHERE fruit <> NULL;
```

!= NULL returns nothing

```
SELECT * FROM fruit WHERE fruit = NULL;
```

== NULL returns nothing

```
SELECT * FROM fruit WHERE fruit LIKE '%';
```

(fruit) returns non-NULL fruit

Person	Fruit
Joseph	Lime
Matt	Apple

progress?

```
SELECT * FROM fruit WHERE fruit NOT LIKE '%';
```

But !(fruit) still returns nothing ∴

Turns out to query NULL data we need the IS NULL & NOT NULL expressions. (special comparators)

```
SELECT * FROM fruit WHERE fruit IS NULL;
```

Person	Fruit
Partha	

of course the best solution is to just use NOT NULL values.

```
SELECT * FROM fruit WHERE fruit IS NOT NULL;
```

Person	Fruit
Joseph	Lime
Matt	Apple

So testing equality when NULL is tricky. How can we join two tables with NULLs in them?

Take the following table:

Person	Fruit
Joseph	Lime
Matt	Apple
Partha	

Fruit	Dish
Apple	Apple crumble
Banana	Banana split
Cherry	
Lime	Daiquiri

There is a operation called NATURAL JOIN. This is a type of Join operation that combines tables based on columns with the same name & data type. It automatically matches & joins the columns that have the same name in both tables. This command gives:

▶ (A `NATURAL JOIN` is like a regular `JOIN` but assumes same named columns ought to be equal).

Person	Fruit	Dish
Joseph	Lime	Daiquiri
Matt	Apple	Apple crumble

So we've lost columns with NULL values. What if we want to include them?

We can use LEFT JOIN & RIGHT JOIN. Technically the JOIN we've seen before is an INNER JOIN.

This is used to basically 'add column values where you can..' & not discard NULL values

For LEFT JOIN, all rows from the left table (the table specified before the LEFT JOIN keywords) are retained ...

For RIGHT JOIN, all rows from the right table (the table specified after the RIGHT JOIN keywords) are retained ...

... regardless of whether a match is found in the other table.

```
SELECT person, fruit.fruit, dish
FROM fruit
LEFT JOIN recipes
ON fruit.fruit = recipes.fruit;
```

(LEFT)

Person	Fruit	Dish
Joseph	Lime	Daiquiri
Matt	Apple	Apple crumble
Partha		

```
SELECT fruit.fruit, dish, person
FROM fruit
RIGHT JOIN recipes
ON fruit.fruit = recipes.fruit;
```

Fruit	Dish	Person
Lime	Daiquiri	Joseph
Apple	Apple crumble	Matt
	Banana split	

```
SELECT recipes.fruit, dish, person
FROM fruit
RIGHT JOIN recipes
ON fruit.fruit = recipes.fruit;
```

(RIGHT)

Fruit	Dish	Person
Lime	Daiquiri	Joseph
Apple	Apple crumble	Matt
Banana	Banana split	
Cherry		

This issue can usually be fixed when combined with NATURAL JOIN (RIGHT/LEFT NATURAL JOIN)

→ Here we had to change the SELECT column as the recipies.fruit column was more complete.

```
SELECT fruit, dish, person
FROM fruit
RIGHT NATURAL JOIN recipes;
```

Fruit	Dish	Person
Lime	Daiquiri	Joseph
Apple	Apple crumble	Matt
Banana	Banana split	
Cherry		

Finally what if you want to do RIGHT JOIN and LEFT JOIN (so no rows are discarded).

For this we use FULL OUTER NATURAL JOIN

```
SELECT *
FROM fruit
FULL OUTER NATURAL JOIN recipes;
```

Person	Fruit	Dish
Joseph	Lime	Daiquiri
Matt	Apple	Apple crumble
Partha		
	Banana	Banana split
	Cherry	

Statistics

In general, don't use SQL for stats. Import the data to a language with a proper statistical analysis toolbox.

Because of this I'll just plonk the slides in here.

In the last lecture we introduced `COUNT` as a way of counting how many things exist?

- How may different fruits are in the outer joined table?

```
SELECT *
FROM fruit
FULL OUTER NATURAL JOIN recipes;
```

Person	Fruit	Dish
Joseph	Lime	Daiquiri
Matt	Apple	Apple crumble
Partha		
	Banana	Banana split
	Cherry	

```
SELECT COUNT(fruit)
FROM fruit
FULL OUTER NATURAL JOIN recipes
```

COUNT(fruit)
4

...So it looks like `COUNT` ignores `NULL`

Lets rank fruits!

Fruit	Stars
Apple	0
Banana	4
Cherry	
Lime	5

```
SELECT AVG(stars) AS Average FROM ranking;
```

Average
3.0

```
SELECT SUM(stars)/COUNT(fruit) AS Average
FROM ranking;
```

Average	(int rounding error)
2	

Remember computers are *awful*

- Multiply count by 1.0 to "*fix*"?
- Also number of stars is *ordinal* data so the *mean* shouldn't be used anyway...

The standard deviation is how far something deviates *on average* from the *mean*.

```
SELECT SQRT(AVG(Deviation)) AS STDDEV
FROM (
    SELECT Fruit, Stars, Mean,
        (Stars-Mean)*(Stars-Mean) AS Deviation
    FROM ranking JOIN (
        SELECT AVG(stars) AS Mean
        FROM ranking
    )
    WHERE stars IS NOT NULL
);
```

STDDEV
2.16024689946929

You can nest queries inside one another (subqueries!)

- This is a recipe for making your SQL *slow*
- Maybe just use SQL for data retrieval and leave complex stats to statistical programming languges?

Lecture 6/6

Mock 1

12/19

Q1) A ✓

Q2) B ✓

Q3) A ✗

Q4) C ✓

Q5) B ✓

Q6) A ✓

Q7) Unassessed

Q8) Unassessed

Q9) C ✓

Q10) D ? ✓

Q11) D ✓

Q12) A ✗

Q13) A ✗

Q14) B ✓

Q15) D ✗

Q16) D ✓

Q17) D ✓

Q18) C ✓

Q19) ✗

Q20) A ✗

Q21) C ✗