Databases

Joseph Hallett

February 19, 2024

What's all this about?

When you write a program all your data disappears after the program ends

- Unless we save it somewhere

SQL Databases are a sensible choice for where to save your data

- Highly optimized storage of tabular data
- Fast and well understood query language
- Fault tolerant protocols

So what is a database?

Super fancy spreadsheet

- Each database will contain *tables* that store data
- Data in tables can be *queried* using a language called SQL
- Data in tables can be *joined* with data in other tables to answer questions

Designing them so you don't tie yourself in knots is tricky!

So why not just use *Spreadsheets*?

- See *Matt Parker's* excellent *Stand-up Maths* video: UK Government loses data because of Excel mistake.

`https://youtu.be/zUp8pkoeMss`

Different types of database

Traditionally, the database would reside on a separate machine

- Space is expensive!
- If you wanted to use the database you had to connect to it

But nowadays space is cheap

- Local per app databases very common

If you need remote data access:

- Use a *server*-style database like *MariaDB or MySQL*

Otherwise use a file-style database:

- ...just use *SQLite*.

Should I use a database?

Am I being paid to store/process this data?

- Yes? Use a database.
- No? Use a spreadsheet (or a database)

Does the data need to be accessed remotely?

- Yes? Use a server-style database (MySQL/MariaDB)
- No? Use a file-style database (SQLite)

Am I just playing with data or is my data tiny (gigabytes in size)?

- Yes? Use a database or plain text data storage (i.e. CSV).

Is my data ***really*** big (petabytes in size)?

- Yes? Use a NoSQL database (beyond scope of this course)

Does my data contain recursive data structures (i.e. lists of lists of arbitrary length)

- Yes? Use *Prolog* or *Datalog*. (or abuse a database `;-))`

Relational Modelling

Databases let us store data in tables!

- But how do you *structure* your data in a table?
- And can we draw pretty doodles based on them?

Relational modelling

Proviso!

Relational modelling is a tool for thinking about how to decompose relationships between things into tables.

- People get fussy about the syntax

Please don't!

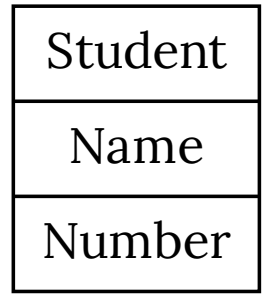
I'll try and show you various syntaxes you may encounter, but its *just a tool*

- Do whatever works for you
- So long as its clear it doesn't matter
- The diagrams are for doodling ideas *not* final implementation

Things are nouns!

Here is a student! Students have a name and a number!

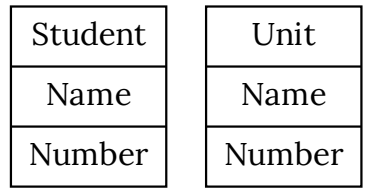
- The student is the *entity*.
- The name and number are the *attributes*.



More things are nouns!

Here is a unit! Units also! have a name and a number!

- The unit is the *entity*.
- The name and number are the *attributes*.



Don’t worry about names

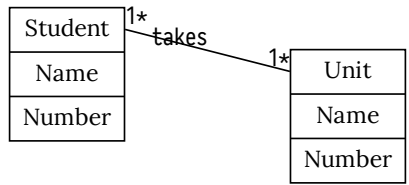
There may be many examples of different ***values*** that could be examples of units and students... but don’t worry about that.

Student	
Name	Patrick McGoohan
Number	6

Unit	
Name	Software Tools
Number	COMS10012

Nouns can be related!

One student may take many units; and units may have many students



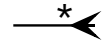
Alternative notation

Some people prefer a graphical notation for entity relationships called *crow's foot*

- I prefer to write it explicitly

Don't get too hung up on notation!

- And use a key if you're ever asked in an exam
- The point is to let *you* doodle notes
- Do whatever makes sense to you or the people you work with



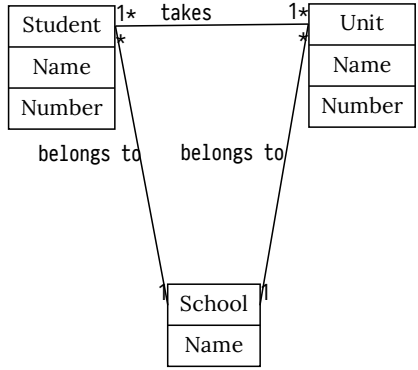
Schools are a thing!

There are things called *schools*:

- Schools have names
- Each unit belongs to *exactly* one school
- Each student belongs to *exactly* one school

Each school can have students and units its responsible for

- But could also be empty!



What should I call a student?

Obviously their name would be *polite*...

...but what will happen if we were to open a class on *Gallifrey*?

All 12!

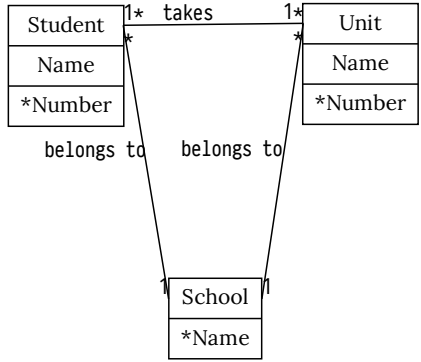
This would rapidly get too confusing for computers!

- (But not for people)

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 uniquely refer to it.
- The *primary key* for an entity is the key we use.

If a key contains multiple attributes its called a *composite key*.
If a key is a meaningless ID column you added just for the sake of having a key its called a *surrogate key*.



When we want to turn it into tables

Every *entity* becomes a *table*

- Each table has a primary key

Every *edge* becomes a table

- Contents of these tables are the *primary keys* of the two items being linked
- Attribute that refers to another key is called a *foreign key*

School Membership

Student	School
6970	School of Computer Science

School Units

Unit	School
COMS10012	School of Computer Science

Student

Name	Number
Joseph Hallett	6970

Unit

Name	Number
Software Tools	COMS10012

School

Name
School of Computer Science

Class Register

Student	Unit
6970	COMS10012

Conclusions

1. Doodle entity relationship diagrams to sketch out database designs
2. Convert to databases by making everything a table
3. Don't get hung up on notation

SQL Basics

We've got a database for storing data...

- It'd be nice to be able to acutally use it and make queries!

For that we need SQL:

- Structured Query Language

SQL

Query language for asking questions about databases from 1974

- Standardized in 1986 in the US and 1987 everywhere else
- Still the dominant language for queries today

Not a general purpose programming language

- Not Turing complete
- Weird English-like syntax

Standardized?

You would be so lucky!

- *In theory*, yes
- *In practice*, absolutely not

Every database engine has *small* differences…

- Some have quite big ones too!

Lots have differences in performance

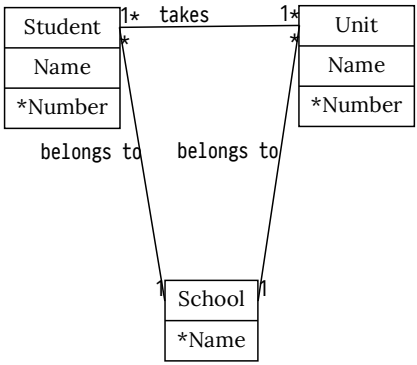
- SQLite is good with strings, most others prefer numbers

Managing these differences used to be an entire degree/job in its own right!

- Now we just manage databases badly!

I'll try and stick to SQLite's syntax…

CREATE TABLE




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

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

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

CREATE 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));
```

Lets build it in SQL

DROP TABLE

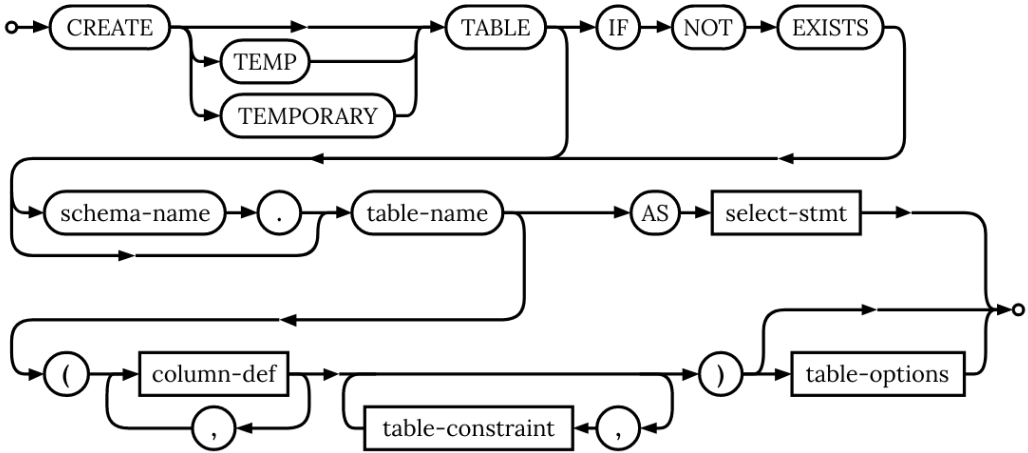
What about if we want to delete them?

```
DROP TABLE IF EXISTS class_register;
DROP TABLE IF EXISTS student;
DROP TABLE IF EXISTS unit;
DROP TABLE IF EXISTS school;
```

Syntax, syntax, syntax

If you go on the *SQLite* documentation page...

- ...you can find syntax diagrams for all of SQL!
- `https://www.sqlite.org/lang_createtable.html`



Types

When creating the fields in our database we made them all of type `TEXT`...

- What other types exist?

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

But really types

Databases sometimes *simplify* these types

- SQLite makes the following tweaks...

`INTEGER`	whole numbers
`REAL`	lossy decimals
`BLOB`	binary data (images/audio/files...)
`VARCHAR(10)`	*actually* `TEXT`
`TEXT`	any old text
`BOOLEAN`	*actually* `INTEGER`
`DATE`	*actually* `TEXT`
`DATETIME`	*actually* `TEXT`

(others may exist... *read the manual!*)

Table constraints

In the earlier examples we marked some columns as `NOT NULL`

- Others as `PRIMARY KEY` and others as `FOREIGN KEY`...
- ...what other constraints have we got

...but SQLite won't actually enforce any of these types or constraints unless you ask it to `:-(`

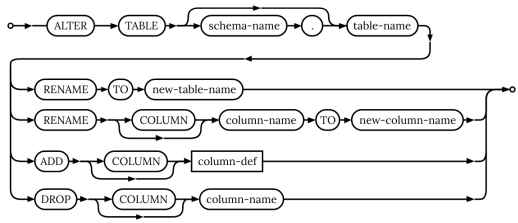
- Check out the `STRICT` keyword when creating the table.

`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

Can I add constraints later?

Yes with the `ALTER TABLE` statement

- But often easiest just to save the table somewhere else
- Drop the table
- Reimport it



What about if we want to add data to a table?

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

So far

We've introduced how to:

- `CREATE TABLE`
- `DROP TABLE`
- `INSERT INTO`

Next step: querying data!

I'm going to use a database from an old iTunes library for demo purposes

- Chinook database

SELECT

Basic command for selecting rows from a table is SELECT

```
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

JOIN

Ideally we'd like those two tables combined into one...

```
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

Reducing the columns...

Clearly there are too many columns here... lets 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

Renaming columns

Title and *Name* aren't particularly meaningful without context

- Lets name them something sensible

```
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

I'm feeling rocky

I want to listen to something a bit rocky...

- Lets filter all the albums to the ones that have *Rock* in the title

```
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

Who rocks?

So who has put out an album with *Rock* in it?

```
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

How many *rock* albums has each artist put out?

Lets `group` by artist and `count` the albums!

```
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

Really we want this list ordered...

Lets `group` by artist and `count` the albums...

- And order it by album count!

```
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
LIMIT 5;
```

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

Basics of SQL

So thats the basics of SQL!

- You can do a ***bunch*** more things with SQL `SELECT` statements...
- ...you can pick them up as you write queries.
- ...most SQL engines have a bunch more counting and query functions too

Go read the documentation!

Normal Forms

Database theory!

- So far we've discussed how to doodle database designs...
- We've discussed how to create tables in SQL

How do we design tables that are easy to use?

Lets start with our records database again...

We could store our data as follows:

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

Please, no.

This is a *terrible* idea

- Yes we have one big table which seems neater
- But its much harder to do anything actually with

For example:

- How many albums does each artist have?
- Change all of *Prince*'s albums after 1993 to being by a *Love Symbol*
- How many artists have an album with the same name?

Normal forms

Normal forms prevent this sort of insanity

- Using them requires discipline, and rememebering rules...
- But is worth it for your sanity in the short to medium term

First Normal Form

Each column shall contain *one* (and only one) value

Each row says describes ***multiple*** albums per artist...

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

First Normal Form

Lets fix that...

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

Lets add some more data to our 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

Second Normal Form

Every non-key attributue is fully dependent on the key

In this case the key is *Artist*, *Album*

- And arguably *year* too if you're gonna pull a Taylor Swift and rerelease all your albums...

Is *Prime Minister* dependent on the key?

- No. Put it in a different table.

Now it looks like

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

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

> *Every non-key attribute must provide a fact about the key, the whole key and nothing but the key; so help me Codd.*

Lets add some extra information to our table of Prime Ministers...

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

Our key is **(*Year*, *Prime Minister*)**; Birthday depends on ***Prime Minister***.

- So every non-key depends on the key...
- So 2NF

But not 3NF as Birthday ***doesn't*** tell you a fact about the ***whole key***... just the Prime Minister.

So split it up!

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

Why is this better?

- Now if we need to alter the birthday of a PM (or any other fact about that key)...
- ...then we only need to alter it in one place.

Other normal forms...

Boyce-Codd Normal Form

A slightly stronger form of 3NF...

- Sometimes called 3.5th Normal Form

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

- Split a 3NF table into tables with single candidate keys to get 3.5NF.

4th Normal Form

If multiple attributes in a table depend on the same key,

- Then those attributes should be dependant too
- Otherwise split them into separate tables...

5th Normal Form

It's in 4th normal form and you can't split it into more separate tables.

This is all getting a bit mathsy...

You can look up formal definitions for each of the normal forms

- (and you should)

But so long as you keep things *as separate as possible*, you'll usually hit at least 3NF by accident.

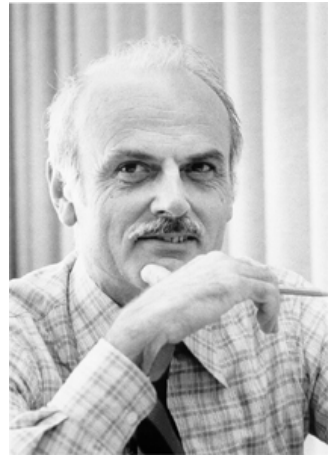
- ...and practically speaking your probably good then
- Getting it to 5NF *does* make things more flexible in the long run...
- But a 3.5NF database is often *good enough*.

Ultimately design is subjective (somewhat).

- ...but mathematical proof of flexibility is good right?

Codd's Law

Every non-key attribute must provide a fact about the key, the whole key and nothing but the key; so help me Codd.



Ted Codd

Lets get back to SQL

So far we've introduced basic SQL

- How to create tables
- How to add and delete data
- How to run basic queries

Lets go further!

- More features, more function
- Other joins
- `NULL`

`NULL` is nothing

There is a special value in SQL to represent missing data: `NULL`.

- But they're ***pretty much always*** a bad idea
- The logic for comparing them is pretty whacky

NULL = NULL?

Lets say we have a database with the following table:

Person	Fruit
Joseph	Lime
Matt	Apple
Manolis	

Lets find everyone who we know what their favourite fruit is!

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

Err..., lets try the opposite?

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

Err what?

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

Person	Fruit
Joseph	Lime
Matt	Apple

So...

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

NULL is weird...

Because NULL means *attribute missing*...

- The results of comparing with it are ~~just plain stupid~~ *somewhat unexpected*

The simple solution is to declare *everything* as NOT NULL

- And use a higher normal form (5NF) then you'll find they almost entirely disappear

Otherwise you have to memorise a bunch of ~~stupid~~ *special* comparators

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

Person	Fruit
Manolis	

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

Person	Fruit
Joseph	Lime
Matt	Apple

Tricky joins

Clearly testing for equality when `NULL` is problematic.

- So what happens when you want to join two tables together with `NULL`'s in them

Person	Fruit
Joseph	Lime
Matt	Apple
Manolis	

Fruit	Dish
Apple	Apple crumble
Banana	Banana split
Cherry	
Lime	Daiquiri

What's my favourite food?

So what might make a nice dish for each of your lecturers?

- (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

But what about poor *Manolis*? How do we get him to appear in our table?

LEFT and RIGHT JOIN

When doing our previous `JOIN` we wanted only rows that matched...

- Technically called an `INNER JOIN`...

Sometimes we're okay with the database sticking `NULL` in if we want to keep columns where a join *can't* be made...

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

Person	Fruit	Dish
Joseph	Lime	Daiquiri
Matt	Apple	Apple crumble
Manolis		

RIGHT JOIN

A `RIGHT JOIN` is like a left join but the other way round...

```
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	

Where has the Banana gone?!

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

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

(Or just `NATURAL JOIN` and it'll *usually* take care of it...)

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

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

One more JOIN!

What if we want to do a LEFT and a RIGHT JOIN at the same time?

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

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

What about statistic functions?

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
Manolis		
	Banana	Banana split
	Cherry	

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

COUNT(fruit)
4

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

Other statistics...

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
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...

What about standard deviation?

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?

So thats SQL!

Tips for using it?

- Don't overcomplicate things!
- Normal forms make things simpler!
- Avoid `NULL` like the plague

JDBC

In the *real world* we rarely want to access a database in its own right

- Rather it is used within a programming language as part of a program

Different languages have different APIs for different databases...

- ...but Java has the *JDBC* for almost all of them

JDBC

- Library is in `java.sql` and `javax.sql` packages
- Wraps all of a databases functionality into something that looks a lot like *Oracle* SQL.
- Supports ***prepared statements*** (you want to use these)

What does it look like?

```
import java.sql.*;

try (final Connection conn = DriverManager.getConnection("jdbc:sqlite:database.db")) {
   conn.createStatement()
         .executeQuery("CREATE TABLE users(username TEXT PRIMARY KEY, password TEXT)");
} catch (final SQLException err) {
   System.out.println(err);
}
```

```
java.sql.SQLException: No suitable driver found for jdbc:sqlite:database.db
```

Lets add some suitable users...

```
import java.sql.*;
import java.util.*;

final var users = new HashMap<String, String>();
users.put("Joseph", "password");
users.put("Matt", "password1");
users.put("Manolis", "12345");

try (final Connection conn = DriverManager.getConnection("jdbc:sqlite:database.db")) {
   conn.createStatement().executeUpdate("DELETE␣FROM␣users");
   final var statement = conn.prepareStatement("INSERT␣INTO␣users␣VALUES(?,␣?)");
   for (final var user : users.keySet()) {
      statement.setString(1, user);
      statement.setString(2, users.get(user));
      statement.executeUpdate();
   }
} catch (final SQLException err) {
   System.out.println(err);
}
```

And list them back out...

```
import java.sql.*;
import java.util.*;

System.out.println("|User␣|␣Password");
try (final Connection conn = DriverManager.getConnection("jdbc:sqlite:database.db")) {
   final var results = conn.createStatement()
      .executeQuery("SELECT␣*␣FROM␣users");
   while (results.next())
      System.out.println("|␣"+results.getString(1)
                    +"␣|␣"+results.getString(2));
} catch (final SQLException err) {
   System.out.println(err);
}
```

Matt	password1
Joseph	password
Manolis	12345

Why not this...

When adding all the users we used a `PreparedStatement` to add all the users.

```
final var statement = conn.prepareStatement("INSERT␣INTO␣users␣VALUES(?,␣?)");
for (final var user : users.keySet()) {
   statement.setString(1, user);
   statement.setString(2, users.get(user));
   statement.executeUpdate();
}
```

Wouldn't this be easier?

```
for (final var user : users.keySet())
   conn.createStatement()
      .executeUpdate("INSERT␣INTO␣users␣"+"VALUES␣('"+user+"',␣'"+users.get(user)+"')");
```

SQL Injection

This leads to a ***horrible*** vulnerability called an ***injection attack***

- You can do something similar with shellscript too `;-)`
- Search for ***Shellshock vulnerability*** if you're interested...

What a prepared statement does is ensure that the things you add are what you say they are
Suppose you do the something similar for the login code:

```
SELECT username FROM users
WHERE username = "Joseph"
AND password = "password";
```

username
Joseph

Suppose the username and password are taken from a website login form...

- What happens if I try and login with a password of:

```
" OR 1 OR password = "heheh
```

Bad things

With a prepared statement:

```
SELECT username FROM users
WHERE username = "Joseph"
AND password = """␣OR␣1␣OR␣password␣=␣""heheh";
```

Without a prepared statement:

```
SELECT username FROM users
WHERE username = "Joseph"
AND password = "" OR 1 OR password = "heheh";
```

username
Matt
Joseph
Manolis

ALWAYS USE PREPARED STATEMENTS

The compiler will even spew warnings and errors about this nowadays...

Transactions

Another *cool* thing that JDBC makes easy are *transactions*...
Suppose you want to do a bunch of additions and updates to a database...

- What happens if something goes wrong *in the middle*?

You *could* go and manually *roll back* all the new data you added and changes you made...

- Sounds tedious
- Lets automate it!

Transaction workflow

1. Start a new transaction
2. Do your work
3. Commit to it when done
4. Rollback if an error occurs

And in Java please?

```
import java.sql.*;
import java.util.*;

try (final Connection conn = DriverManager.getConnection("jdbc:sqlite:database.db")) {
   conn.setAutoCommit(false);
   final var save = conn.setSavepoint();
   try {
      conn.createStatement() .executeQuery("INSERT␣INTO␣users␣VALUES␣('Alice',␣'pa55w0rd')");
      conn.createStatement() .executeQuery("INSERT␣INTO␣users␣VALUES␣('Bob',␣'Pa55w0Rd7')");
      if (true) throw new Exception("Whoops!");
      conn.createStatement() .executeQuery("INSERT␣INTO␣users␣VALUES␣('Eve',␣'backd00r')");
      conn.commit();
   } catch (final Exception err) {
      conn.rollback(save);
   } finally {
      conn.setAutoCommit(true);
   }
} catch (final SQLException err) {
   System.out.println(err);
}
```

Now if we query users...

```
SELECT * FROM users;
```

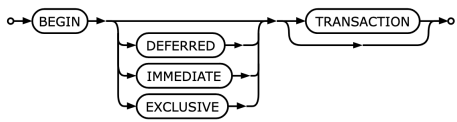
username	password
Matt	password1
Joseph	password
Manolis	12345

username	password
Matt	password1
Joseph	password
Manolis	12345

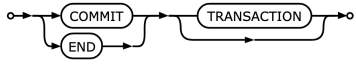
Our table *remains* unaltered... the whole transaction was *rolled back*.

(Oh, and BTW SQLite also can do transactions in SQL)

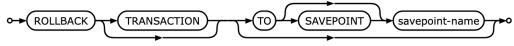
begin-stmt: hide



commit-stmt: hide



rollback-stmt: hide



Conclusions

JDBC lets you access SQL from Java

- Make sure you load the right driver
- Catch SQLExceptions
- Use prepared statements and transactions to prevent errors
- And an ORM like *Hibernate* if you like.

IMPORTANT NOTE

Please don't actually implement password storage like we did in the lecture...

- Go speak to someone in the cyber or crypto groups *first*...
- Or read NIST 800-63 first

I will write papers about you if you do `;-)`

> *Joseph Hallett, Nikhil Patnaik, Benjamin Shreeve and Awais Rashid. "Do this! Do that!, And nothing will happen" Do specifications lead to securely stored passwords? 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). 2021.*