Databases and the Cloud: The Notes

Josh Felmeden

2018
December

Contents

1 The Internet

End systems are connected via the **communication links** that consist of the different types of physical media. Usually, the end systems are not directly attached by a single link, but rather they are attached through a router.

There are two kinds of host: *clients* and *servers*. A program or machine that responds to request and others is called a **server** while a program or machine that sends the requests to the server is called a **client**.

The internet is made possible by the development, testing, and implementation of the *internet standards*. They are developed by the Internet Engineering Task Force (or the IETF). Their documents are known as RFCs (request for comments). There are a number of protocols, such as TCP, IP, HTTP, and SMTP (this one is used for emails). There are more than 2000 RFCs.

1.1 Protocols

A **protocol** is a set of rules that govern the communication to ensure a standard of communication. It also consists of messages sent and actions taken in response to replies or other such events.

A simple protocol could be where one machine sends a message (called a *request*) and another machine replies with a response. This can then be repeated.

1.2 Internet Layers

- HTTP
  - Makes request
  - Reads and handles the response
- TCP
  - Breaks data up into packets
  - Puts the packets back in order and reassembles messages
- IP
  - Attaches to and from addresses to each packet
  - Reads and groups packets based on the address
- Physical internet
  - Send bits to local routers

  - Receives bits and assembles into packets

1.3 HTTP: Hyper text transfer protocol

What's the difference between the web and the internet? Well, the internet is the computer network itself (or the whole infrastructure): while the web (or the world wide web) is an application that runs on that infrastructure.

It's probably the most common application protocol that there is on the web (but there are others like video streaming and FTP and the like). Right now, there's a version 2.0, but we'll be focussing on version 1.1 here.

1.4 Crud

CRUD is an acronym for the basic operations that can be carried out on data.

- Create
  - The create interaction creates a new resource in a server assigned location. The create interaction is performed by a HTTP POST method.
- Read
  - The read interaction accesses the current contents of a resource. The interaction is performed by a HTTP GET method
- Update
  - The update interaction makes a whole new version for an existing resource (or makes a new one if there isn't one)
- Delete
  - The delete interaction deletes an existing resource

1.5 Structure

HTTP is *line-based* and each line ends with a **carriage return line feed** (CR LF). In it, there is a header and a method.

1.6 Status codes

There are some cases where an interaction does not go well. The response from a server can be a number, and the first digit informs you of the nature of the error.

1.7 URLs

The internet needs to have addresses. It needs to know the addresses of both the client and the server. The URL (**uniform resource locator**) tells you where some resource is. A resource is an *address*.

2 Developing web pages

2.1 Markup

Historically, marking up a paper manuscript was done by editors to show authors how to revise their manuscripts. The markup was done in *blue pen* to make it distinguishable from the manuscript text.

In electronic documents, **tags** are used to make the markup distinguishable from the content. A markup language is used to annotate a document.

2.2 HTML

HTML (or hypertext markup language) consists of a fixed set of *tags* that describe how information should be displayed. For example:

```
<p> This is some text </p>
<h1> This is some header </h1>
```

The Browsers do not display the HTML tags, but they use them to render the content of the page.

HTML5 is different from HTML because it's simpler, but also **semantic** (which means that some of the tags describe what the data means as well has how it should be displayed). It also has some more features.

Example HTML5:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>My title</title>
    </head>
    <body>
        content
    </body>
</html>
```

Tags have to be nested too.

A block-level element always starts on a new line and takes up the full width available. Conversely, an inline element doesn't start on a new line and it only takes up as much width as is necessary.

The <div> tag is a block level tag that has no specific meaning. This is OK to use for layout purposes, but you should not use it as a replacement for something that should be a semantic tag. Because the semantic tags are mostly a new addition to HTML5, older frameworks used <div> all over the place to structure the pages.

2.2.1 Attributes

In this example:

```
<p id="today">
    28 September
</p>
<p class="info">
    lecture 2
</p>
<p class="info">
    QB 0.18
</p>
```

2.2.2 Links

Almost anything can go inside a <a> tag: text, images, other HTML elements. The href could be a full URL, or it can be relative to the current page.

The main issue with HTML is that you need to structure your web pages really carefully because it's going to be viewed on all kinds of devices and browsers.

Here are some basic rules:

1. Use lower case element names
2. Close all your elements (you don't need to close them in HTML5 but do it anyway)
3. In HTML5, it's optional to close the empty statements, but do it anyway.
4. HTML5 allows the mixing of uppercase and lowercase names, but just use lowercase because it looks nicer and it's easier to write.
5. HTML5 allows attribute values without quotes but again, it's bad because it looks ugly
6. ALWAYS add the `alt` attribute to images, because if, for some reason, the image can't be displayed, you need some alternate text to display. It's also used for people using screen readers.
7. In HTML5, the `html` and `body` tag can be omitted, but, again, it's **bad**. It can crash some XML software.
8. To ensure that everything is interpreted and has correct search engine indexing, the language AND the character encoding should be defined as early as possible.

9. Don’t use absolute pixel width measurements

2.3 Forms

The form tag is used for things like buttons, text boxes, etc. It has input types of things like:

- Button
- Month
- Number
- Text
- Password
- Color
- Date
- ...

The **action** attribute defines the action to be performed when the form is submitted. Normally, the data from the form is sent to a web page on the server when the user clicks on the submit button. For example:

```
<form method="post"
action="/action_page.php">
</form>
```

In this example, the data is sent to a page on the server called `"/action_page.php"`. This page contains a script that will handle the form data such as storing it in a database.

There are two (2) methods to send form data, **GET** and **POST**. In HTML5, browser forms support them both. GET places form data in the URL parameters by default (GET/search?query=pancakes), while POST sends the data in the HTTP request body. There are fewer limitations and it’s more secure because the data is not visible in the URL.

2.3.1 Validation in forms

If you use `type="number"`, then it won’t let you type in letters. It you use `required`, then the browser won’t let you submit if the field is empty.

Place holder is text that can be displayed while the field is empty. It’s NOT a label.

2.3.2 Buttons

Buttons can have these types:

- Submit (default)
- Reset: reset all form fields
- Button: do nothing by default (use this if you're using JS).

2.4 CSS

Use HTML for the structure, and then CSS for the styling. This includes layout, appearance, and some behaviours. You can customise ANYTHING. If you want emphasised words to be underlined, then by golly you can do that.

CSS stands for Cascading Style Sheets. CSS describes how HTML elements are to be displayed on screen. CSS saves a lot of work. It can control the layout of a number of multiple web pages all at once. It can be added to HTML elements in 3 ways

- **Inline** which means that it uses the style attribute in HTML elements
- **Internal** which means tat it uses a `<style>` element in the `<head>` section.
- **External** which means that it links to a CSS file. This is the recommended method btw.

Linking to a stylesheet looks something like this:

```
<link rel="stylesheet" href="styles.css">
```

This goes in the head tag of any web page. External style sheets have a few advantages, namely that a single style sheet can control a lot of pages. In general, you have a number of different pages that share a common style. You can define the style sheet in a single document and then have all of the HTML files refer to the same CSS file. It also facilitates *global changes* because if you're using external styles, you make a change in one place and then it's automatically propagated to all of the pages in the system. Finally, it allows for the *separation* of content and design. With the external CSS, all of the design is housed in the CSS and the data is in HTML.

Oh, also note that there are two kinds of reference: absolute (for example the absolute reference of a page like `href="http://www.example.com/theme.css"`) while the relative looks like `href="/themes/theme.css"`.

Here's an example of a style sheet:

```
h1 {
    font-family: sans-serif;
}

.lecture {
    font-weight: bold;
}

em { font-style: normal; }
em.room { font-style: italic; }
```

2.5 Data Formats and Operations

In 2015, Bristol elected a councillor for each of the 24 wards. Here are the results for this:

Candidate	Party	Votes
Chris Davies	Liberal Democrat	2435
Christopher Louis Orlik	Labour	1499
Glenn Royston Vowles	Green	722
Claire Lisa Louise Hayes	Uk Independence	625
Anthony Paul Lee	Conservative	590
Domenico Hill	Trade Unionists and Socialist Coalition	37

Here's what the structure would look like if it was written in C:

```
struct Candidate {
    char  name[100];
    char party[100];
};

struct Ward {
    char name[100];
    int electorate;
};

struct Result {
    struct Candidate candidate;
    struct Ward ward;
    int votes;
};
```

Here's how it would look in Java:

```java
class Candidate {
    String  name;
    String party;
};

class Ward {
    String name;
    int electorate;
};

class Result {
    Candidate candidate;
    Ward ward;
    int votes;
};
```

And here's how we use the data structures

```java
winner.candidate.name = "Chris Davies";
winner.candidate.party = "Liberal Democrat";
winner.ward.name = "Knowle";
winner.ward.electorate = 8820;
winner.votes = 2435;

System.out.println("In " + winner.ward.name + ", the winner was " +
winner.candidate.name + " with " + winner.votes + " votes.");

>> "In Knowle, the winner was Chris Davies with 8820 votes."
```

The question now is: How do we read data from storage? Well, what we shouldn't do is create our own data format (unless you're a big boy like Google/Microsoft). What we should do is to use the existing standards for encoding and storing data.

2.5.1 UNICODE and Encoding

Every file is a sequence of bytes that can further be broken down into bits. What the bytes mean are entirely dependent on how the are encoded and what sort of file type it is. 1 byte means that there are 256 possibilities. This is more than enough for most Western languages, but the stupid Chinese have something like 50k characters depending on how you count them. We also might want to encode more than the alphanumeric characters like icons, emojis, line drawing chars, mathematical symbols and so on.

ASCII was invented in the early 60's as a standard character set for computers and electronic devices. It's a 7-bit set containing 128 chars. It contains all of the English alphanumeric characters and some special characters. Unfortunately, once a lot more countries got involved in the internet, we needed a few more characters to allow us to represent their languages and alphabets. This is where Unicode came in.

The Unicode Consortium develops their Unicode standard. It replaces the existing character sets (ASCII) with a new one that is then implemented in many language such as HTML, XML, Java, and even the Latex I'm writing this in. Unicode defines a crazy 136k characters. **Encoding** is how these numbers are translated into binary numbers to be stored and processed in a computer. There are different ways that Unicode characters or code points can be encoded. There are some fixed width and some variable length encodings.

Line endings are kind of complicated, because in UNIX based devices, the ending is written as 'LF', or, in binary, '0000 1010'. However, in early mac, it's written as 'CR', or, in binary, '0000 1101'. Some systems even use a combination of the two.

Some network protocols have fixed conventions. For example, HTTP uses ASCII and defines a line ending as CRLF.

2.5.2 Tables and CSV

The simplest 'structured' file format is a list file with one entry per line. We can read or write to this in a loop. *Tables* are for 2-dimensional data. Another option we have is a CSV, or **comma separated values**. This might look like:

```
Candidate, Party, Votes
Chris Davies, Liberal Democrat, 2435
...
```

2.5.3 Stream Processing

In a stream, data items arrive one at a time and you only get to see them once each. We can use these streams if processing can be done with a single pass over the data, or if we only need to access recent data. We cannot do stream processing if we nee to do multiple passes through a data set or we need random access to items in the data set.

There are a few stream operations: filter, map, and reduce.

Filter means the we only read certain values that matches some condition.

Maps apply a function to each of the data items that are received through a stream.

Reduce applies a function to the whole stream and then output a single output.

We can also chain these operations.

Streams are really important because when we develop data driven web apps, we often need to process these streams of data, and if we bear these in mind when we write the code, we can structure it accordingly. *Map* and *filter* are stateless, per-element tasks. They are easy to parallelise. Some *reduce* operations can be done in parallel too.

2.6 Representing Data as Trees

If a list is 1-dimensional, and a table is 2-dimensional, what on earth is a 3-dimensional data? Actually, it's pretty easy, we just pick a third separator character.

Tree structure (or tree diagram) is a way of representing the hierarchical structure of data. It's called tree structure because it resembles a tree (duh doi) even though the diagram is generally upside down compared to a tree. The top most level is called the **root** while the bottom most ones are called the **leaves**.

Interestingly, most data can be represented as a tree. If we look at the candidate class from above:

```
<candidate>
    <name>Catherine Slade</name>
    <party>
        <name>Green</name>
    </party>
    <ward>
        <name>Bedminster></name>
        <electorate>9951</electorate>
    </ward>
</candidate>
```

Another way to represent this might be:

```
{
    "name":"Catherine Slade",
    "party": {"name:": "Green"},
    "ward": {"name": "Bedminster", "electorate": 9951},
}
```

Escape characters are used to signify that a character sequence needs to get special treatment from the same characters. Here are some now:

- \n is a line feed
- \r is a carriage return
- \\ is a back slash
- \" is a double quote.

2.6.1 XML

XML is pretty straightforward to use all over the internet. It's also easy to write programs that process the XML documents.

HTML is all about displaying the information, while XML **describes** information. XML is the most common tool for data manipulation and data transmission. It can also be used for data storage. XML is both human AND machine readable, while also being flexible enough to support platform and architecture independent data interchange. XML allows a software engineer to create a vocabulary and use it to describe data (also sometime called being an **extensible** language).

The properties of XML include:

- Information identification
- Information storage
- Portable and non-proprietary
- Data transfer

The components of XML are:

- The declaration
- The root element
- Attributes
- Child elements
- Text data

In XML there are different steps for validation and processing. There are a total of 2 validation methods. The first is called DTD, or **Document Type Definition**. The other is called **schema**.

Here is an example of how this works;

Example XML:

```
<candidate>
    <name>Catherine Slade</name>
    <party>
        <name>Green</name>
    </party>
    <ward>
        <name>Bedminster></name>
        <electorate>9951</electorate>
    </ward>
</candidate>
```

DTD validation:

```
<?xml version="1.0"?>
<!DOCTYPE candidate [
<!ELEMENT candidate (name, party, ward)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT party (name)>
<!ELEMENT ward (name, electorate)>
<!ELEMENT electorate (#PCDATA)>
]>
<candidate> ... </candidate>
```

XML schema are another way of describing and XML document structure in XML itself. Nuts, right? Schemas are *more powerful* than DTDs. Also, an **XSD** is an **XML schema definition**.

Example schema:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="candidate">
        <xs:complexType> <xs:sequence>
    <xs:element name="name"type="xs:string"/>
    <xs:element name="party"> <xs:complexType> <xs:sequence>
        <xs:element name="name"type="xs:string"/>
    </xs:sequence> </xs:complexType> </xs:element>
        <xs:element name="ward"> <xs:complexType> <xs:sequence>
            <xs:element name="name "type="xs:string"/>
            <xs:element name="electorate"
                        type="xs:nonNegativeInteger"/>
        </xs:sequence> </xs:complexType> </xs:element>
    </xs:sequence> </xs:complexType> </xs:element>
</xs:schema>
```

XML entities are kinda like escape sequences, and they're also used in HTML. For example, there are:

- $<$ is written by & lt;
- $>$ is written by & gt;
- " is written by & quot;

XPath/XQuery are ways in which to address nodes in an XML document. This also works for non-XML versions of HTML and can be very useful in web applications. They're kinda like CSS selectors but have a different syntax.

The final thing to worry about with XML is the XSLT. This stupid acronym means *eXtensible Stylesheet Transformation Language* and it's used for transforming XML documents into other documents, such as HTML, PDFs, and even back into XML documents.

2.6.2 JSON

JSON (or JavaScript Object Notation) is a widely used data format for web apps. It is text, so it can be used to exchange data between a server and it's client browser. On top of that, it's also easily human readable and writable. Because of the fact that it's text, it can be parsed and generated by the majority of programming languages. Neato!

JSON syntax is derived from JavaScript syntax, such as the Data is in the format of name/value pairs. Also, data is separated by commas. Curly braces hold objects, and square brackets hold arrays.

Here's an example of some JSON:

```
{
    "name:" "David",
    "id": 101,
    "units":
    ["COMS10010", "COMS2XXXX"]
}
```

JSON is cooler than XML in some aspects because:

1. JSON doesn't use end tags
2. JSON is shorter
3. JSON is quicker to read and write
4. JSON can use arrays
5. And probably the biggest difference is that XML has to be parsed with an XML parser, while JSON can be parsed by a standard JS function, or in Java by the GSON library.

Gson is cool in it's own right, but it's not entirely relevant and I'm lazy, so I'll summarise to say that it's Google's answer to JSON.

2.6.3 Templating

Templating is a method in which we can provide a template for a sentence and then just fill in the necessary blanks with the required words. For example:

```
{
    "id":2,
    "name":"Bedminster",
    "electorate":9951,
    "x":135,
    "y":225
}


<p> The ${name} ward had an electorate of ${electorate} in ... </p>
```

The output of this would be: 'The *Bedminster* ward had an electorate of *9951* 9951 in ... '.

Free marker is a templating library, and it is a Java library that generates text output (such as HTML web pages, e-mails, etc) based on the templates and changing data. Templates are written in a simple, specialised language called 'FreeMarker Template Language' (FTL). A general purpose programming language such as Java is used to prepare the data, and then Apache FreeMarker displays this data using the templates.

Here's an example of some FreeMarker code:

```
<#include "header"> <!--include other templates-->
<#if name??> <!-- condidionally display the elements-->
    <p> Hello, ${name}</p>
</if>
<#list messages as m>
    <p>${m.type}: ${m.text}</p>
</#list> <!--create some output for each element in a list-->
```

And this is the extent of the Web! Hope you enjoyed! xoxo

3 Relational Databases

In databases, we have a three tier architecture setup. Firstly, we have the **client**, which is the presentation layer. This is normally made of things like HTML, CSS, JavaScript etc. After this, we have the Application server. This is the **business logic layer**, and is normally coded in Java. Finally, we have the database server, called the data services layer. This is usually managed with a **DMS** (or database management system).

Databases are really good because although CSVs are good for storing small amounts of data that doesn't need to change often, it doesn't really scale up. If there are large amounts of data that frequently change, AND it needs to have it's integrity at all times, they're not suited to the job.

An example of this is with Bristol University. Say we had to write a program to count the number of books with a title containing the word 'research'. Also suppose that all of the book records are stored in a text file on a disk. How long would it take us to get through it?

If us, as a mere human, were to go through this, it would probably take us about 7 months to get through the lot of it (if it wasn't ordered). So, we need some algorithm and data structure for applications that want to store such ridiculously large data structures. And database systems are just the answer we're looking for.

Let's look at another example. Bank accounts need to store all of the customers bank balances in a file on a disk. So we can write a program to handle the ATM transactions, such as 'debit £30 from account 024858'. If, for some reason, it's a shared account, and I withdraw 10 pounds from the account 1 second before my friend (who wants to withdraw £100), then the following events would occur:

1. The bank would read the balance from my request and return £1000
2. The bank would read the balance from my friend's request and also return £1000
3. 1000 is more than 10, and also more than 100, so the transaction can continue.
4. My friend is closer to the account, so his ATM writes the balance back to the bank first, with the updated balance being 900.
5. My ATM performs the operation 1000-10 to be the new balance, and writes this back to the

bank.

6. My balance is now 990 despite me having taken out a total of 110 pounds

This is obviously not something that we want to be happening on a regular occasion, so we need some special protocols to be implemented if the data is going to be manipulated by multiple users concurrently. Again, database systems are the answer to this.

The final example we are going to look at is when we bank transfer someone. If we take a simple program as an example, when we transfer £100 to someone, for a split second, the money has vanished, because it has been taken out of my account, and is going to be transferred into my friend's. If the system were to crash at this precise moment, what would happen to the "vanished" money? Data with strong integrity requirements should probably be managed by a database.

3.1 Tables

A table in a database might look something like this:

house	street	town	postcode
3	Merton Street	Oxford	OX1 4JD
22	Ambrose Street	York	YO1 3PQ
3a	Victoria Road	Malvern	WR14 1UB
21	Woodland Road	Bristol	BS8 1UB
23	Woodland Road	Bristol	BS8 1UB

The **schema** for this table would be 'Address(house, street, town, postcode).

3.1.1 Keys

To address the data in the table, we use something called *keys*. We have two different kinds of key in databases:

- **Superkey** – a combination of fields that uniquely determines a row.
  - This means that after choosing data for the fields in the superkey, we have *no choice* over the rest of the data in that row.
  - In the table above, '{house, postcode}' would be a superkey.
  - If we fix data for *house* and *postcode*, then street and town are determined, so there's no choice for this data.
  - {countryName} would be a superkey for the schema 'Countries(countryName, capitalCity, continent)', while {continent} would not be.

- **Candidate key** – also just called a key. It's a superkey that's also minimal.
  - This means that if you remove any field from a key, it ceases to be a superkey.
  - For example, in the address schema, {house, postcode} is a key, while {postcode}, {house, postcode, street} is NOT a key.

3.2 SQL

Here are some quick facts about the SQL language:

- It is not necessary to populate all fields (depending on the table definition of course).
- Strings are 'single quoted'.
- Table constraints are a way of asking the DBMS nicely to guarantee the *integrity* of your data.

3.2.1 Constraints

Constraints are useful because your DBMS doesn't know anything about the thing you're writing your database about, so it can't know what needs to be really strictly monitored. This is what a constraint looks like:

```
CONSTRAINT key-constr UNIQUE (name, street, town)
```

The 'key-constr' is a name to be used in error messages, and the 'unique' part tells the DBMS what a key is. For example, if we run this:

```
INSERT INTO BankBranches
    VALUES ('HSBC', 'Queens St', Bristol, '22-11-12');
INSERT INTO BankBranches
    VALUES ('HSBC', 'Queens St', 'Bristol', '22-12-16');
```

Then we'd get an error because there's a duplicate entry for the key.

Each table needs to have *exactly* one **primary key**. Other than this, constraints are an opportunity for you to provide some more information to the DBMS, and therefore it will be able to tell you when you're being a complete dumbass.

You also can't drop tables that a table has some foreign key to, because then the DBMS will get angry (if you set up the constraints right - again, stops you from being stupid).

3.3 Projection Selection and Null

This handy little table will show you the kind of similarities there are between relational programming concepts and the imperative programming ones:

Imperative Programming Concepts	Relational Programming Concepts
Assignment	Projection
Sequencing	Selection
Conditional branching	Product
Bounded iteration	Join
Unbounded iteration	Aggregation

3.3.1 Projection

If we had this table:

x	y	z
1	a	yes
2	b	no
3	c	no
4	d	yes

If we were to do ‘project y,x’, we’d end up with:

y	x
a	1
b	2
c	3
d	4

We could also do something cool like “project x+1, ‘hello’" to get:

x+1	hello
2	hello
3	hello
4	hello
5	hello

3.3.2 Selection

If we were to perform “Select z = ‘Yes’", we’d get:

x	y	z
1	a	yes
2	b	no
3	c	no
4	d	yes

$\xRightarrow{\text{select z = ‘yes’}}$

x	y	z
1	a	yes
4	d	yes

We could also do something a bit fancy with selection like:

x	y	z
1	a	yes
2	b	no
3	c	no
4	d	yes

$\xRightarrow{\text{select z = ‘yes’} \wedge x>2}$

x	y	z
4	d	yes

3.3.3 Combining them together

With the following table:

id	title	cp
1	Databases	10
2	Web Technologies	10
3	Types of λ-Calculus	10
4	Overview of Computer Architecture	20
5	Programming in C	30

And we want to build a table containing only the titles of units that have 20 credit points or more, we can do the following:

```
SELECT title FROM Unit WHERE cp >= 20
```

This would give us the necessary results.

Here are some more fancy instructions:

CWMarks

student	CW1	CW2
1	60	73
2	28	54
3	72	70

```
SELECT student, CW1*0.6 + CW*0.4 AS average
    FROM CWMarks
    WHERE CW1*0.6 + CW2*0.4 >= 40
```

Results in:

student	average
1	65.2
3	71.2

3.3.4 NULL data

SQL treats NULL data as an absence of data, or a 'I don't know'. Typically, built in functions and operators return NULL when *any* of their inputs are NULL. For example:

- `3*NULL = NULL`
- `CONCAT('hello',NULL)= NULL`
- `NULL > 8 = NULL`

However, there are a couple of exceptions:

- `NULL is NULL = 1`
- `NULL IS NOT NULL = 0`

In general, always know if your data can be null, and if it cannot, then declare the field as `NOT NULL`. If your data *can* be null, consider the `IS NULL` case in your selections.

3.4 Product and Join

Let's say that there are two databases, unit and lecturer. If we wanted to know who the unit director is for the databases unit, we currently don't have a way of knowing this. So, what we do is, we can take the **Cartesian product** of the two tables, which means that we get all of the possible combinations of data, and after this, we remove the ones that don't make sense.

There are two ways we can do this. Looking at the following example:

Mage

id	name	mage	spell
1	Harry	1	1
1	Harry	1	2
2	Hermione	2	1

Spell

id	spell
1	Expelliarmus
2	Lumos

If we do the Join of Mage and Spell on id = Mage, then we get the following:

id	name	mage	spell	id	name
1	Harry	1	1	1	Expelliarmus
~~1~~	~~Harry~~	~~1~~	~~1~~	~~2~~	~~Lumos~~
~~1~~	~~Harry~~	~~1~~	~~2~~	~~1~~	~~Expelliarmus~~
1	Harry	1	1	2	Lumos
2	Hermione	2	1	1	Expelliarmus
~~2~~	~~Hermione~~	~~2~~	~~1~~	~~2~~	~~Lumos~~

The `Join` keyword glues tables together by using their keys (normally).

3.4.1 Select-From-Where

This query is an incredibly useful query, and forms as a skeleton for all other queries. For example, if we wanted to find all of the lecturers in the same group as anther person, when can do a 'select from where' query on tables that are joined together:

```
SELECT R.name
    FROM Lecturer L JOIN Lecturer R ON L.rgroup = R.rgroup
    WHERE L.name = 'Peter'
```

3.4.2 Join variants

There are a few variants of the join command, and we'll go over them here.

- **Natural join** joins the tables on their *common columns*.
- **Left join** is another kind of join, but I don't really know what it does.
- **Right join**
- **Inner join** is the normal join that we already know
- **Outer join** is called the full outer join
- **Cross join** is written 'table1, table2'

3.4.3 Set operations

- `Query1 UNION ALL Query2` also called the *bag union*
- `Query1 UNION Query2`
- `Query1 INTERSECT Query2`
- `Query1 EXCEPT Query2`

Oh yeah don't forget about *entity relationship diagrams* where the * means that it's a primary key, and there are primary and foreign key relationships.

3.5 Aggregation and nested queries

Imagine we have a table of all the students in a course, and we want to know the average grade of each of the students over all the units:

Enrol

Student	Unit	Grade
200	100	60
200	101	50
201	100	70
201	101	60
201	102	80

How would we do this?

The first thing we can do is group by a certain value, for example the student ID. But, what happens to the other values, that change? Well, this is where aggregation comes in. We have a few options for what to do with this list of values. For example, we can take an average, take the maximum, etc. What we **can't** do is to have more cells overflowing. We can also take two different columns as the grouping factor.

Adding to our trust 'Select from where' skeleton, we can now add `GROUP BY`, and now, we get something looking like this:

```
SELECT columns
FROM table
WHERE condition
GROUP BY keycolumn
```

It's important in SQL to list the key column in the columns if you want it to appear in the output. For example, going back to the first table, if we perform `SELECT student, AVG(grade)AS average FROM enrol GROUP BY student`, we'd get:

Student	Average
200	55
201	70

Here's a cool list of all of a lot of the aggregation functions:

- `MIN()`
- `MAX()`
- `AVG()`
- `SUM()`
- `COUNT()`
- `COUNT(DISTINCT)`
- `COUNT(*)`

To avoid problems, each column specifier in the SELECT part should return **at most** one value when it's evaluated on a group. This is *guaranteed* if each column is either mentioned in the GROUP BY clause, an aggregate function application, or a constant.

Another thing we can add is the `HAVING` keyword. If we want to select from some table with 2 conditions, then this 'having' keyword allows us to use a second condition to further expand our query. For example:

```
SELECT columns
    FROM table
    WHERE condition1
    GROUP BY key_column
    HAVING condition2
```

Oh, did you think we were done? NO, there's also the `ORDER BY` key word which, weirdly, orders the results. We cal also add on the DESC or the ASC filter to sort the results in descending or ascending order respectively. And there's a `LIMIT` keyword, that limits the results by the number you give it.

3.5.1 Nested Queries

In the following table, if we wanted to find all the lecturers in the same research group as 'Peter', how would we do it?

Lecturer

ID	Name	Rgroup
1	David	null
2	Steven	80
3	Janet	83
4	Nick	80

The query would look something like this:

```
SELECT name
FROM Lecturer
WHERE rgroup = (
    SELECT rgroup
    FROM Lecturer
    WHERE name = 'Peter'
)
```

Another thing we could do is to find the titles of the units whose director is in the hardware group:

```
SELECT title
FROM Unit
WHERE director IN (
    SELECT id
    FROM Lecturer JOIN Rgroup
        ON Lecturer.rgroup = Rgroup.id
    WHERE Rgroup.name = 'Hardware'
)
```

There are even more cool things that we can do with nested queries, like get the deviation from he student's overall average for each unit grade:

```
SELECT Enrol.student, unit, grade - T.average
    FROM (
        SELECT student, AVG(grade) AS average
        FROM Enrol
        GROUP BY student
    ) AS T /* new table needs to be named */
    JOIN Enrol ON Enrol.student = T.student
```

As an additional note, if you're going to aggregate over the whole table, don't use GROUP BY, use something like:

```
SELECT COUNT(name) FROM student
```

3.6 Normalisation

Oh boy! Normalisation! This is the process where your tables get into shape. Tables that are in normal form are no longer plagued by certain kinds of redundancy (bad) and dependency (bad). These things can cause all kinds of anomalies when inserting, updating and deleting data.

3.6.1 Functional dependencies

Cities in the UK

City	c_pop	Region	r_pop	Country
Bristol	0.44M	SW	5.2M	England
Bath	88.8K	SW	5.2M	England
Manchester	0.52M	NW	7M	England

If you do this, you're an idiot. Why would you keep all of these region and region data separate, when you could just reference it in another table? You lend yourself vulnerable to **update anomalies** because of the population of the SW region increases, you gotta update it in a load of tables.

Also, you might get **insert anomalies**, which is because we can't have a city that's not in a region, or have a region with no cities.

Delete anomalies are also bad because if we remove the last city in a region, the region would cease to exist.

By definition, a functional dependency is an attribute (A) that depends on a set of attributes (XS), just if the value of the attribute is uniquely determined after fixing set attributes. We write the dependency as $XS \to A$.

Look at the following table for an example:

Lecturer

*uname	fname	lname
csxdb	David	Bernhard
csxds	David	Smith

We have the following dependencies:

- {uname} $\rightarrow$ fname
- {uname} $\rightarrow$ lname
- {uname, fname} $\rightarrow$ fname
- {uname} $\rightarrow$ uname
- {fname} $\not\rightarrow$ lname

Some of these are *trivial* because they are already in the set they are said to be functionally dependent with.

Fixing these dependencies isn't too hard. If we know all of the functional dependencies, we just pick them out, and slap them in another table with a foreign key.

3.6.2 First normal form

This is an example of a terrible table:

Bad table

name	username	units
"David"	"csxdb"	"COMSM0016, COMS10010"
"Alice"	"csxaw"	""
"John"	"csxjs"	"COMS20002"

Let's go through the normalisation steps.

Lecturers

name	*username
David	csxdb
Alice	csxaw
John	csxjs

Units

*unit	director
COMSM0016	csxdb
COMS10010	csxdb
COMS20002	csxjs

Here, the username is the foreign key for the director part of the Lecturers table.

3.6.3 Second normal form

Definition: **key attribute**

An attribute is said to be a *key attribute* if it is part of some (candidate key), otherwise it's a *non-key attribute*

For a database to be in 2NF, we need:

- The database to be in 1NF
- There are no dependencies where XS → A where A is a non-key attribute and
- XS is all key attributes but is not a superkey

This looks like:

*house	*postcode	city
MVB	BS8 1UB	Bristol
Flat D.01	BS8 4UN	Bristol
Flat D.02	BS8 4UN	Bristol
House of Commons	SW1 0AA	London
30	EC3A 8BF	London

$\downarrow$ to 2NF

*house	*postcode
MVB	BS8 1UB
Flat D.01	BS8 4UN
Flat D.02	BS8 4UN
House of Commons	SW1 0AA
30	EC3A 8BF

*postcode	city
BS8 1UB	Bristol
BS8 4UN	Bristol
SW1 0AA	London
EC3A 8BF	London

3.6.4 Third normal form and transitive dependencies

Transitive dependencies are where you have something like $A \to B \to C$. For example, in this table:

*postcode	city	region
BS8 1UB	Bristol	SW
BS8 4UN	Bristol	SW
SW1 0AA	London	L
EC3A 8BF	London	L

We have the dependency {postcode} $\rightarrow$ {city} $\rightarrow$ region. Therefore, for a schema to be in 3NF we need:

- The schema to be in 2NF
- The schema to have no *non-trivial* dependencies $XS \rightarrow A$ with:
  - A is a non-key attribute
  - XS contains a non-key attribute, but not a superkey

From the table above, we end up with these two tables:

***postcode**	**city**
BS8 1UB	Bristol
BS8 4UN	Bristol
SW1 0AA	London
EC3A 8BF	London

***city**	**region**
Bristol	SW
London	L

3.6.5 BCNF - Boyce Codd Normal Form (3.5NF)

Just when you thought we had enough normal forms, here's another one. The problem we currently have is the following:

student	**email**	**society**
bb16801	bilbo@...	Cheese and Wine
bb16801	bilbo@...	Hoverboard
mm16280	mickey@...	Hoverboard
cl16343	calvin@...	Pantosoc
ht16991	hobbes@...	Pantosoc

The functional dependencies are:

- {student} $\rightarrow$ {studnet} which is trivial.
- {student, society} $\rightarrow$ {email} (LHS superkey)
- {student} $\rightarrow$ {email} which is the problem

A sensible schema that's in 3NF is normally also in BCNF because you need some effort to not be. Basically, 3NF is in BCNF automatically unless it contains at least two composite candidate keys that overlap.

Every schema can be decomposed to 3NF in a way that is both *lossless* and *dependency preserving* (which means that no functional dependencies are removed). For BCNF and higher, this is no longer the case. In general, we should normalise to 3NF unless there is a really really good reason not to. Then, continue to BCNF if it's possible and sensible.

3.7 JDBC: SQL in Java

Picture this: we have a login system with a username and a password box. The SQL might look something like this (if we enter 'Gandalf' for username and 'Mellon' for password):

```
SELECT id FROM Users WHERE name = 'Gandalf' AND pass = 'Mellon';
```

We can do something called **preparing a statement** so that we are able to pass in the text from the textboxes. This is what a prepared statement looks like:

```
stmt = prepare (
    "SELECT id FROM users WHERE name = ? AND pass = ?"
);

result = execute(stmt,
    ["Gandalf", "Mellon"]);
```

A Java database API is called a JDBC. An implementation comes with the Java installation. Its classes are in the java.sql and javax.sql packages.

To connect, we type the following:

```
import java.sql.Connection;
import java.sql.DriverManager;

// in a function
Connection c = DriverManager.getConnection(
    CONNECTION_STRING
);

// do stuff
c.close();
```

All JDBC methods throw the checked exception 'java.sql.SQLException':

```
try {
    // database stuff
} catch (SQLException e) {
    // handle or bail
}
```

The result looks like this:

```
PreparedStatement s = c.prepareStatement(
    "SELECT id, email FROM Users" +
    "WHERE name = ? AND pass = ?"
);

ResultSet r = s.executeQuery();
while (r.next()) {
    String email = r.getString("email");
    id = r.getInt("id");
    // do somthing
}
s.close();
```

And finally:

```
try {
    c = openconnection();
    doStuff(c);
} catch (SQLException e) {
    //handle exception
} finally {
    try {
        if (c != null) { c.close(); }
    } catch (SQLException e) {
        // handle exception
    }
}
```

3.7.1 Transaction

Transaction: a sequence of one or more operations on the database that must be executed as one.

ACID

- **Atomicity:** Either all the operations in the transaction are completed or none of them are.
- **Consistency:** If the database was consistent before the transaction, it is consistent afterwards.
- **Isolation:** Transactions execute independently. If two transactions complete at the same time, then it should be the same result if they were to operate one after the other
- **Durability:** Once the transaction is committed, the effects should not be lost through some later failure.

Most of the DBMSs have a transaction manager that takes care of the ACID properties so we don't need to worry about it. Normally, this is achieved through some form of *locking* or *timestamps*. For durability, *transaction logs* are kept.

By default, every statement runs in its own transaction unless it is not autocommitted.

4 Security

4.1 Web Security

Rule number 1 of security is that all data coming from the client is assumed to be malicious until you've properly validated it.

4.1.1 Sessions and cookies

HTTP is *stateless* which means that you can send a request, get a response, and then the connection ends. If you send another request, the server doesn't and can't know that it's coming from the same person.

Cookies come in because then we get some state between requests. If a response contains a **cookie** header, then all of the further requests to the same server will include this header.

Session hijacking is when you break into someone's account by stealing a copy of their session data. This can be made harder by not using predictable session tokens, using TLS encryption, or expiring the inactive sessions.

A **session token** is a cryptographically random (about 128-bit entropy) that isn't linked to the username or password. It might look like: 4d58f7a-5ff6-4523-a344-f8125381dc9c.

What you *shouldn't* do is to include a password or other secret data in a cookie. Not even in encrypted/encoded format. Also never authenticate people by checking if their name appears in the cookie because anyone can manually set 'COOKIE=admin' in their browser.

To get a secure cookie, there are two things we can set:

- **HttpOnly** means that the cookie is not accessible by JavaScript which means that it is protected against scripting attacks.
- **Secure** means that the cookie will only be sent for TLS encrypted requests.

We should always try to use these unless you're not sure if you're using TLS in which case just leave off the "secure" bit.

JSON web tokens were supposed to be an alternative to cookies for non-browser clients such as apps and APIs, but they're really easy to get wrong and terrible, terrible things have happened with these in the past.

4.1.2 SOP, CORS, XSS, CSRF

XSS: Cross-Site Scripting An XSS attack looks like:

```
<script>alert("OH NO");</script>
```

Whenever you show user-generated content (such as a blog or forum post), remember rule number 1. You don't want people to be able to write their own JS to be able to alter the page view or even worse (such as stealing data of the person viewing the page).

One of the ways we can avoid this is to HTML escape any user-generated content that you display in a HTML page, or offer them some library that they can use (like markdown).

Another defence is to use HTTP security headers, where, if it's set, only the whitelisted scripts are allowed to execute.

CSRF: Cross-Site request forgery Let's say that you are logged into some forum on your computer as well as some random bank. On the forum, someone says 'Click this link!', and the code is:

```
<a href="bank.example.org/transfer?destination=alice&amount=100">link</a>
```

You click on one site that causes another action on another site that you're logged into. Most of th responsibility lies with the target site (the bank in this case) if it allows this kind of thing to happen but as a user you can help by logging out of the important sites when you're not using them.

Some of the defences you can employ are to check the referrer and the origin of the HTTP headers. If necessary, use the per-request CSRF security tokens (more on this in a second). You can also re-authenticate before really important operations (such as changing a password). For an API, you can also require custom 'X-' headers.

CSRF tokens are a one-time token that is a hidden field in a form.

SOP and CORS SOP, or **Same Origin Policy**, mean that browsers only allow JS requests to the same origin (this means the same protocol, host, port, etc) as the source of the script.

CORS, or **Cross Origin Resource Sharing** , is where you set a header in your responses to sites that are using open data from some resource that you're hosting. You whitelist certain domains that are allowed.

4.1.3 TLS: Transport Layer Security

Let's say that we want to encrypt your traffic with a key so that anyone that's listening doesn't know what's going on. The problem with this is that an attacker could actually be on the path to the real server, which is actually more likely than some rando listening in.

Okay, so this might not work, another idea that might work is that browsers have root certificates from CAs built in. Servers buy certificates from CAs, so this seems okay. Let's encrypt is a free CA by Mozilla that should be good enough for most sites. Always try and use TLS if you have a domain name of your own.

4.1.4 Password Storage

If you're a genuine idiot, then you'll store passwords in plain text. Just don't do this. It's just as bad to store them in base64, or any form of encoding because this adds 0 to the security of the password. Hashing is also bad because if two people pick the same password, then they get the same hash.

What you should do is to pick a random *salt* for each user and store the *hash* of the salt and the password so that it doesn't show up that users have the same passwords. You can also iterate the hash (or *stretch*) to slow down the brute force attacks.

In general, just use a library. Don’t implement your own crypto without a PhD or you’ll have a bad time.

4.2 Database security

The most dangerous thing that is posed to a database is an SQL injection. With this tool, many accounts can be breached because the data can be dumped out of the database. Also, leaks happen in the weirdest places, like VTech was breached because people just don’t really give it a decent password. Also, there was almost 0 protection on the data. The passwords were hashed with MD5, which is actual trash.

4.2.1 SQL injection

An SQL injection happens where the data from a form is interpolated into a string, and then the whole string is parsed as SQL. Because the user could actually provide SQL code in the form, this becomes dangerous. An example of bad code is:

```
execute("SELECT * FROM Member " +
        "WHERE login_name = '" + name +
        "' AND password = '" + pw + "'")
```

If we supply the name as ‘David’, and the password as `' OR '' = '` then the equivalent SQL statement produced would be:

```
SELECT * FROM Member
WHERE login_name = 'David'
AND password = '' OR '' = ''
```

Since `'' = ''` always evaluates to true, we can just get the results from this database (it will return everything). The way we can fix this is to have prepared statements:

```
s = prepare("SELECT * FROM Member " +
            "WHERE login_name = ? " +
            "AND password = ?")
s.execute(name,pw)
```

This eliminates the risk because the data is never parsed as SQL.

One way that people think will fix it is to backslash escape (insert before ’ and). This is because user supplied data can contain the escape characters so that the user data gets parsed as SQL.

4.2.2 Direct Object reference

The my account page for a website could look something like ‘www.kidspass.co.uk/accounts/*ID*’. You could just change the account ID in the URL and see someone elses account (something that ha happened to online banking services too in previous years).

An example could be that your website has three style sheets (red.css, green.css, and blue.css) in the folder `/var/www/styles`. Your server responds to ‘GET /page?colour=red.css’ with the page and then the contents of ‘/var/www/ + colour’.

But, what if we request `GET /page?colour=../../etc/hosts`? This would get us `/var/www/../../etc/hosts = /etc/hosts`. The host file often reveals the internal network configuration. Other interesting files are `/etc/passwd` which contains user account information. This attack gets any file on the system that the web server has access to. Of course, we could solve this by sanitizing the colour parameter, but the root problem is that it’s a direct reference to a filename.

A way to solve it would be to make it an indirect reference. The way we do this would be to have the request as `GET /page?colour=1`, and then the SQL would be `SELECT file FROM Styles WHERE id = ?` and then have a table like:

id	file
1	red.css
2	green.css
3	blue.css

This indirect reference means that the user can only pass an id that the application tries to match against the styles table to get a filename. An invalid id cannot fetch a file that’s not in the table. Of course, you don’t have to use a database for this, but if you do protect it against an SQL injection. Compared to sanitizing the filename, it’s a bit harder because you have to register all valid files in the database.

Another example is that you run a bank website. When customers log in, they’re redirected to `https://acme-bank.example.com/user/userid`. What happens if someone edits the user id in the URL bar? The vulnerability is that anything that is sent by the browser can be modified by the user.

Handling a `GET /account/id` request could look like:

```
execute("SELECT * FROM Account " +
        "WHERE id = ?", id);
execute("SELECT * FROM Account " +
        "WHERE id = ? AND owner = ?",
        id, currentUser());
```

Now, the currently logged in userid is used in the query, so that a user is not able to refer to someone else’s account.

4.2.3 Session keys

Session keys are the maximum security possible, but it's high effort. Whenever a user logs in, it finds all the objects that they have access to and then create a temporary session key for each one. Only ever accept the session keys as references. Again, you can't reference someone else's account. This is an indirect reference.

A **business key** is something that your users can understand (such as a username etc) and it's good to keep business keys and primary keys (from the database) separate. If security measures require it, hide the database keys from users. The security gain is that you need someone's database key to refer to their objects (accounts, documents etc.) which is different to their username, which you can get by social engineering for example. This also solves the problem of people wanting to chang their username and people complaining because their automatically assigned customer number ends with 666 or other such ridiculousness.

Whenever a user tries to access an object, check whether the user is allowed to access this object. The workflow for `GET /account/id` is:

1. Sanitize id
2. Get the subject (that's currently logged in)
3. Check whether this subject is allowed to access this object
4. Log the request if necessary
5. Return the object if it's allowed.

Sanitizing the id means that we reject things like 'aaa' if it's supposed to be a number (on top of the standard SQL protection).

5 The Cloud

Cloud computing is really good because:

- Faster speed to market
- Optimization of resources
- Increased operational efficiency

Cloud computing is also able to scale along both axes. *Vertical scaling* means that we just get a bigger server. This technique is good because we don't need to change the code. Unfortunately, it's costly and also servers can't grow indefinitely. Conversely, *horizontal scaling* means that we just get lots of small servers. You only get an incremental cost, which is nice, but unfortunately, we now have a distributed system.

5.1 AWS

AWS is a really cool service that allows us to do loads of different things. In the example of hosting a wordpress blog, we use Amazon Route 53 for the DNS, and a single Amazon EC2 instance with full stack on the host. This has its risks, though, because it has a single point of failure. This method is suitable for 1 user.

If we wanted something with more than 100 users, this wouldn't work, so we can separate the single host into the web host, and the database host. We then use a manage database (RDS). With this service, there's no failover, which could prove to be bad.

If we wanted some even bigger service, we would need it to be highly available. We could have 2 web services that are in different availability zones with elastic load balancing. This is still not perfect because it's got moderate performance but low efficiency.

Finally, if we wanted a big infrastructure (such as about 100,000 users), then we just shift some load around, such as static content to S3 storage and cloudfront CDN.

5.1.1 Autoscaling

Autoscaling is important because sometimes shit blows up (such as black Friday). For this, it adds and removes servers based on load, which means that it's way more cost effective. It also increases elasticity and resilience of service.

5.2 Something as a Service (XaaS)

The three things that we can have as a service are:

- Infrastructure (IaaS) - Rent virtual machines directly
- Platform (PaaS) - Rent machines and OS and basic SW stack (add your own applications)
- Software (SaaS) - rent software and run in the cloud

It's good because it has a lower management overhead. You also don't need to buy or maintain your own hardware and security updates are managed for you. **Performance and reliability** mean that the data centres are less likely to fail than the cables to your own buildings. It's also really **scalable**, and most startups wouldn't be able to exist without this.