Data Management

Matthew Barnes

Contents

Unix fundamentals

Philosophy

Definition of an OS

- Operating system:
    - A program that ties the hardware and the software together.
    - It enables the computer hardware to communicate and operate with the software.
    - It also manages access to the computer's CPU, memory and storage.

OS fundamentals

- The operating system has three fundamentals:
    - Multi-user
    - Multi-processing
    - Multi-tasking

Multi-user

- An OS is multi-user when it can have multiple people using it at the same time.
    - Example: ECS linux server

Multi-processing

- An OS is multi-processing when it can use more than one processor/core.
- Processors can be:
    - Tightly coupled (e.g. in a high-end server)
    - Loosely coupled (e.g. in a cluster of computer nodes)

Multi-tasking

- An OS is multi-tasking when it can handle multiple processes at the same time.
- Example:
    - Windows, Linux, Mac OS etc. are all examples of multi-tasking OS'.
    - MS-DOS versions below 4 cannot multitask.

Philosophy

- Unix set out these norms in software development:
    - Each program should do one thing well instead of lots of things adequately.
    - The output of one program should be the input of another
    - Software should be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
    - Use tools available to help with tasks.

Open-source software

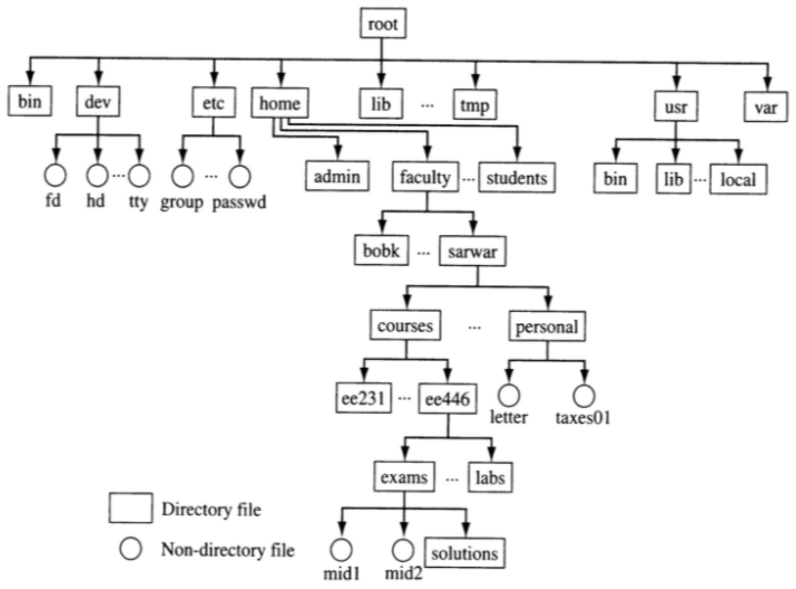
- Publicly released source code you can study, change, and distribute.
- Linux is open-source.

Unix workings

Unix file systems

Navigating the Unix file system

- A typical file system in Unix looks like this:



-
- With 'root' being a parent of everything (similar to C:\ in Windows).
- A few interesting folders to note are:
    - **/etc** stores config files for the system
    - **/var/log** stores log files for various system programs
    - **/bin** and **/usr/bin** stores several commonly used programs

Types of paths

- There are **absolute** paths and **relative** paths.

Absolute paths

- Absolute paths are relative to the root folder (/).
- They always start with a slash
- You can be in any directory and the absolute path of another directory will remain the same.
- They're longer than relative paths.
- Example:
    - /usr/bin/file

Relative paths

- Relative paths are relative to the directory you're in.
- They don't start with a slash

- If you change directory, then a relative path's meaning is changed.
- They can be a lot shorter than absolute paths.
- Example:
    - java/bin (assuming we're in a directory where path 'java/bin' exists)

Path shortcuts

- ~
    - The tilde (~) refers to the home directory of the current user.
    - Absolute path of ~ would be /home/[username]

- .
    - The full-stop (.) refers to the current working directory.
    - Example:
        - If you were in /usr/bin, . would refer to /usr/bin

- ..
    - Two full-stops (..) refers to the parent directory.
    - Example:
        - If you were in /usr/bin, .. would refer to /usr

Hidden files

- If a file starts with '.' it is deemed as 'hidden'.
- This means it won't show up with a normal 'ls'.
- Example:
    - File named "hideme" will be seen with 'ls'
    - File named ".hideme" will not be seen with 'ls'

Useful commands

ls

- 'ls' is used to list files in your current working directory.
- You can use a flag like this: 'ls -t' to alter ls' output slightly.
- Flags:

-l	Long list
-t	Sort by modification time
-S	Sort by size
-h	File sizes in human readable format (e.g. bytes)
-r	Reverse list
-a	Show all files (including hidden files)

- Example:
    - 'ls' will show a normal list of files.
    - 'ls -l' will show a long detailed list of files.
    - 'ls -la' will show a long detailed list of all files, including hidden ones.

cd

- ‘cd’ is used to go to another directory; to ‘**c**hange **d**irectory’.
- The argument is the path we want to move to. If no argument is given, it goes to /home.
- Example:
  - ‘cd’ goes to /home.
  - ‘cd ./java’ goes in the ‘java’ directory in our current directory. The ‘./’at the start can also be omitted as you can see in the next example.
  - ‘cd potato’ goes to the ‘potato’ directory in our current directory
  - ‘cd /usr/bin’ goes to the absolute path /usr/bin

file

- ‘file’ is used to display what type of file something is.
- In Windows, a file type is determined by its extension (.txt, .avi, .mp3 etc.)
- In Linux, you use the ‘file’ command.
- Example:
  - ‘file ./textdoc’ will output information about ‘textdoc’ (located in the current directory) being a text file.

man, info

- The ‘man’ and ‘info’ commands are used when you require information about a certain command.
- If ‘man’ is not enough, ‘info’ will go into more detail.
- Not sure what command to use? Use ‘man -k <search item>’ to search for suitable commands to use.
- Example:
  - ‘man ls’ will tell you about the ‘ls’ command, what it does and how to use it.
  - ‘man -k list directory’ will tell you commands like ‘dir’, ‘ls’ and ‘vdir’.

touch

- The ‘touch’ command is used to create a new empty file.
- Example:
  - ‘touch ./file’ will make a new file called ‘file’ in the current directory.

mkdir

- The ‘mkdir’ command creates a new directory. ‘**mak**e **dir**ectory’
- The argument given is the name of the path to make.
- Using the -p flag will create parent directories, too
- Example:
  - ‘mkdir ./folder’ will make a directory called ‘folder’ in the current directory.
  - ‘mkdir -p ./folder1/folder2’ will make ‘folder1’ and ‘folder2’ inside that directory.

rmdir

- the ‘rmdir’ command will delete a directory. ‘**rem**ove **dir**ectory’
- The directory has to be empty.
- Example:
    - ‘rmdir ./folder’ will remove the ‘folder’ directory in the current directory if it’s empty.

rm, cp, mv

- The ‘rm’ command will **rem**ove a file.
- The ‘cp’ command will **co**p**y** a file to a destination.
- The ‘mv’ command will **mov**e a file to a destination.
- Using the ‘-r’ flag will remove/copy/move a non-empty directory.
- Example:
    - ‘rm ./file’ will remove the file ‘file’ in the current directory.
    - ‘rm -r ./dir’ will remove the entirety of the ‘dir’ directory in the current directory.
    - ‘cp -r ./folder1/folder2 ./folder3’ will copy ‘folder2’ in ‘folder1’ to ‘folder3’ in the current directory.
    - ‘mv ./file1 ./file2’ will simply move ‘file1’ to ‘file2’ (basically renaming) in the current directory.

Create non-empty file

- You can use any of these editors to create non-empty files:
    - Nano
    - Vim
    - Emacs
    - Pico
- You can use this syntax to work on a file:
    - ‘nano ./file’ (edit ‘file’ located in the current directory)
    - The nano program will open and you will be editing the file ‘file’.

Display file

- You can display a file with the following:
    - [“cat” or “more” or “less” or “head” or “tail”] [filename]
- The commands are different ways of displaying a file.
- The argument is the file to display.
- Commands:
    - Cat
        - Dumps entire contents
        - (if you like cat, I suggest checking out bat)
    - More
        - Reads a file a screen at a time, and usually doesn’t allow backward scrolling.
    - Less

    - Displays file, allows forward/backward scrolling. Does not read entire file at start.
  - Head
    - Displays top part of file. Default is 10 lines, but ‘-n’ flag can change this.
    - Example: ‘head -n50 ./file’ will display 50 lines.
  - Tail
    - Same as ‘head’, but the last lines are read instead.

Wildcards

- A wildcard is a way of referencing many files at once.
- There are many kinds of wildcards:
  - *
    - 0 or more characters
  - ?
    - 1 character
  - [ ]
    - Range of characters
    - [abcde]
      - 1 character listed
    - [a-e]
      - 1 character in the range
    - [!abcde]
      - 1 character that’s not in that list
    - [!a-e]
      - 1 character that’s not in that range
  - {debian, linux}
    - 1 word that’s in the list
- Example:
  - *.??? will reference any file with a filename and a 3-letter file extension:
    - hello.txt
    - music.mp3
    - document.doc
  - track[1-9] will reference any file with ‘track’ as a name, followed by a number.
    - track1
    - track2
    - track3 etc.

Permissions

- The ‘chmod’ command will change the permissions of a file
- The ‘ls -l’ command will display the permissions of files

Permission strings

- The permissions (or ‘mode’) of a file can be represented with a 10-character string:
  - _ _ _ _ _ _ _ _ _ _

- File type
    - '-' is regular file
    - 'd' is directory
- Owner access
    - 'r' allows reading
    - 'w' allows writing
    - 'x' allows executing
- Group access
    - 'r' allows reading
    - 'w' allows writing
    - 'x' allows executing
- Other access
    - 'r' allows reading
    - 'w' allows writing
    - 'x' allows executing

- Example:
    - drwxr-xr-x
    - 'd' means that this mode refers to a directory
    - The next 'rwx' means that the owner has full access
    - The next 'r-x' means groups can only read and execute
    - The next 'r-x' means everything else can only read and execute

Permission numbers

- To get the permission numbers of a file, order out a table like this:

Owner			Group			Other		
r	w	x	r	w	x	r	w	x

- If a permission is enabled, put a '1' under it. If a permission is disabled, put a '0' under it.

Owner			Group			Other		
r	w	x	r	w	x	r	w	x
1	1	1	1	0	0	1	0	0

- Now, take the three groups of 1's and 0's:

- 111 100 100
- Convert it from binary to decimal:
- 7 4 4
- You have now represented the permissions of this file in a decimal number.
- Here is a table of each combination of owner/group/other permission:

Binary	Decimal	Meaning
000	0	No permissions
001	1	Execute only
010	2	Write only
011	3	Write + Execute
100	4	Read only
101	5	Read + Execute
110	6	Read + Write
111	7	All permissions

Changing permissions

- You can change permissions using ‘chmod’
- The way arguments are formatted are as follows:
- chmod [person accessing] [operation] [type of permission] [path]
- Example:
    - chmod g+w file
    - Means that we are granting groups the permission to write to the file ‘file’.
    - chmod u+x runme.sh
    - Means that we are granting the owner (ourselves) permission to execute the file ‘runme.sh’.
- All the argument types can be found here:

Person accessing	Operation	Type of permission
‘u’ - user	‘+’ - grant	‘r’ = read
‘g’ - group	‘-’ - revoke	‘w’ = write
‘o’ - others	‘=’ - set	‘x’ = execute
‘a’ - all		‘-’ = none

- You can also change permissions using a number, like this:
    - chmod 744 test.sh

Unix pipes, processes and filters

- Everything in Unix is either a file or a process.
- Every process has a unique ID, known as a ‘PID’.
- ‘init’ is the first process when Unix boots up. All other processes are ‘descendants’.

Useful commands

- ps
    - View processes that are running
- top
    - View CPU usage of all processes
- kill
    - Terminate process with PID argument
    - There are two ways to kill a process:
        - SIGTERM - request termination
            - Usage: ‘kill -s SIGTERM 1447’
        - SIGKILL - force termination
            - Usage: ‘kill -s SIGKILL 1447’
- CTRL-Z
    - Pause foreground process
- bg
    - Move paused process to background
- fg
    - Bring process back to foreground

Running a process when logged off

- You can use the screen window manager to keep programs running even when logged off.
- Commands:

Command	**Meaning**
screen	Create a new screen
screen -d	Detach from existing screen
screen -list	List available screens
screen -r <id>	Resume screen given ID
CTRL-D	Kill screen

Piping

- There are 3 file descriptors for processes:
    - STDIN - Input from the user
    - STDOUT - Normal output
    - STDERR - Error output
- **Piping** refers to redirecting the output of one program into the input of another program.

- Piping a program's output into another program:
    - program_1 | program_2
        - program_1's output goes into program_2

- Pushing a program's output into a file:
    - program_1 > file.txt
        - The output of program_1 will go into file.txt
    - program_1 >> file.txt
        - The output of program_1 will be appended to file.txt

- Get input from a file to put into a program:
    - program_1 < input.txt
        - program_1 gets input from input.txt

Filter

- A filter is a program that takes input, changes it in some way, then outputs it again.
- They can be used just like in piping.
- Some filters:

Filter	**Function of filter**
head	Takes first 10 lines
tail	Takes last 10 lines
sort	Organises data
wc	Prints number of newlines
uniq	Remove duplicate lines
du	Estimate file space usage
xargs	Build + execute command lines

Redirection

- Using ‘>’ sends both STDOUT and STDERR to a file.
- By using ‘1>’ and ‘2>’, you can separate STDOUT and STDERR.
- ‘1>’ sends STDOUT output to a file.
- ‘2>’ sends STDERR output to a file.
- Example:
    - ‘helloworld 1> output.txt 2> error.txt’
    - output.txt will have ‘Hello World’ in it
    - error.txt should have nothing in it, as long as nothing goes wrong

More useful commands

Command	**Function of command**
find	Search for files in directory hierarchy
tar	Create file archive / extract
gzip/gunzip & zip/unzip	Compress files
nohup	Run command in background
parallel	Run jobs in parallel
basename	Removes parent folders from pathname e.g. basename /home/jsmith = jsmith
cut	Extract sections from each line of input
w3m	Text-based WWW browser

Scripting

grep

- The ‘grep’ command will find regex matches in a text and print every line which contains at least one match.
- It goes like this:
- `grep [options] pattern [file...]`
- The ‘i’ flag makes it case-insensitive.
- The ‘c’ flag returns number of matches, instead of the matches themselves.
- The ‘o’ flag returns only the parts of the line that match (instead of the whole line)
- Example:
    - Find any dictionary words with ‘biscuit’ in them:
    - `grep -i biscuit /usr/share/dict/words`
    - Find number of instances where ‘Electronics’ shows up in ‘index.html’:
    - `grep -c Electronics index.html`

sed

- The ‘sed’ command takes in a stream of text and returns it.
- It goes like this:
- `sed [option] file`
- Example:
    - Convert ‘abc’ to ‘cba’ in a file named ‘file’:
    - `sed ‘s/abc/cba/g’ ./file`
        - The ‘s’ means ‘substitute’.
        - The ‘abc’ is the regex to replace
        - The ‘cba’ is the string to replace it with
        - The ‘g’ means ‘global’ (referring to regex)

awk

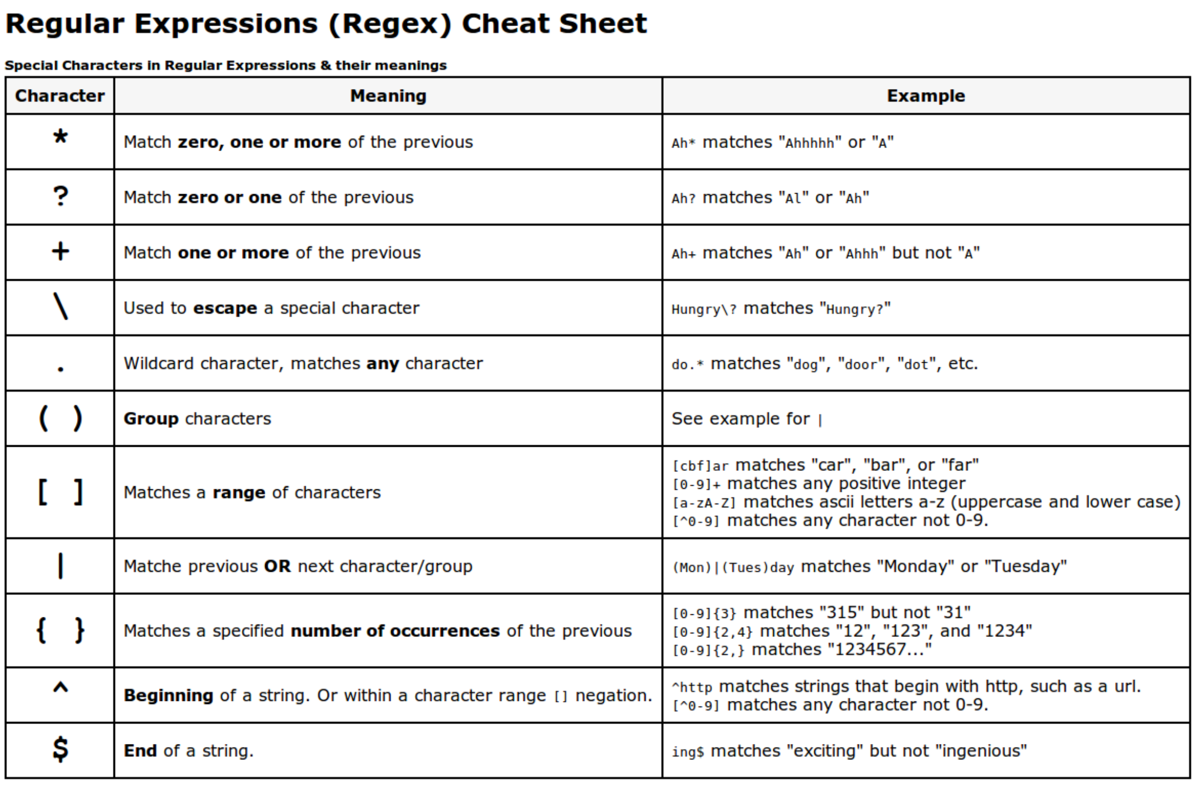
- The ‘awk’ command takes in data and performs some code on it in the AWK programming language.
- AWK is a high-level language with arrays, loops, variables etc.
- AWK is also simple and easy to learn; you can look up anything you need to do online.
- Example:
    - Print ‘hello world’:
    - `awk "BEGIN { print \"Hello, world!!\" }"`
    - Print first and third columns only
    - `awk ' {print $1,$3} '`

Regular expression

- A regular expression (regex) is a sequence of characters that define a search query.

- You can just have normal strings, like this:
    - “fox” => “The quick brown fox jumped over the lazy dog”
- You can also have special characters, like ‘.’ which means any character:
    - “An.” => “Any, Ane, Ank”
- You can also have anchors, like ‘^’ which means ‘beginning of text’:
    - “^A” => “A, A, A, A”
- Or ‘$’ which means ‘end of text’:
    - “t$” => “White goat”
- You can also have quantifiers, like ‘*’ which means ‘0 or more of’
    - “.*” => “This selects everything”
- Or ‘+’ which means ‘1 or more of’ or ‘?’ which means the quantifier before will match as few characters as possible (lazy):
    - “<.+?>” => “<b> <ba> <bar>”

More regex syntax:



Regular Expressions (Regex) Cheat Sheet

Special Characters in Regular Expressions & their meanings

Character	Meaning	Example
*	Match **zero, one or more** of the previous	`Ah*` matches "`Ahhhhh`" or "`A`"
?	Match **zero or one** of the previous	`Ah?` matches "`Al`" or "`Ah`"
+	Match **one or more** of the previous	`Ah+` matches "`Ah`" or "`Ahhh`" but not "`A`"
\	Used to **escape** a special character	`Hungry\?` matches "`Hungry?`"
.	Wildcard character, matches **any** character	`do.*` matches "`dog`", "`door`", "`dot`", etc.
( )	**Group** characters	See example for `|`
[ ]	Matches a **range** of characters	`[cbf]ar` matches "car", "bar", or "far" `[0-9]+` matches any positive integer `[a-zA-Z]` matches ascii letters a-z (uppercase and lower case) `[^0-9]` matches any character not 0-9.
|	Matche previous **OR** next character/group	`(Mon)|(Tues)day` matches "Monday" or "Tuesday"
{ }	Matches a specified **number of occurrences** of the previous	`[0-9]{3}` matches "315" but not "31" `[0-9]{2,4}` matches "12", "123", and "1234" `[0-9]{2,}` matches "1234567..."
^	**Beginning** of a string. Or within a character range `[]` negation.	`^http` matches strings that begin with http, such as a url. `[^0-9]` matches any character not 0-9.
$	**End** of a string.	`ing$` matches "exciting" but not "ingenious"

Data exchange

Network protocol stack

- In the TCP/IP protocol, there are 4 layers.
- Each layer is responsible for a task. This abstracts the parts of TCP/IP, making it easier for different standards to be established.

Layer	**Explanation**	**Examples**
Application	Protocols that software uses to communicate over networks	HTTP (documents), FTP (files), SMTP (sending mail), POP (receiving mail)
Transport	Protocols that manage how the data is sent to the recipient	TCP (establishing a connection, make sure that each system receives messages), UDP (fire and forget)
Internet	Protocols that manage how the packets are structured and distributed	IP, ICMP, IGMP
Link	Protocols concerning the physical hardware used for sending data	DSL, Ethernet, ARP

HTTP web protocol

Requests

- Each time a user communicates with a site, or if a site communicates with another site, it uses a HTTP method. There are a variety of different HTTP methods used over the Internet:

Method	**Meaning**
GET	Request for a web page or object from a server
POST	Send data/information about client to server
PUT	Send document to server
DELETE	Delete object on server
HEAD	Request information about page/document

TRACE	Trace proxies or tunnels in path from client to server
OPTION	Determines server capabilities

- Each time to type a URL into your browser and get a website up, you've sent a GET request.
- Each time you've logged into a website, you should've sent a POST request.

Elements of an HTTP response

- Session: request/response transaction (is this a request or a response?)
- Request: [GET/POST/PUT etc.] + header + [body]
- Response: status header + [body]

Stateless

- Server gets request, performs action, and generates a response.
- Stateless means:
    - Previous interactions have no effect on server
    - Same request results in the same action

Data exchange formats

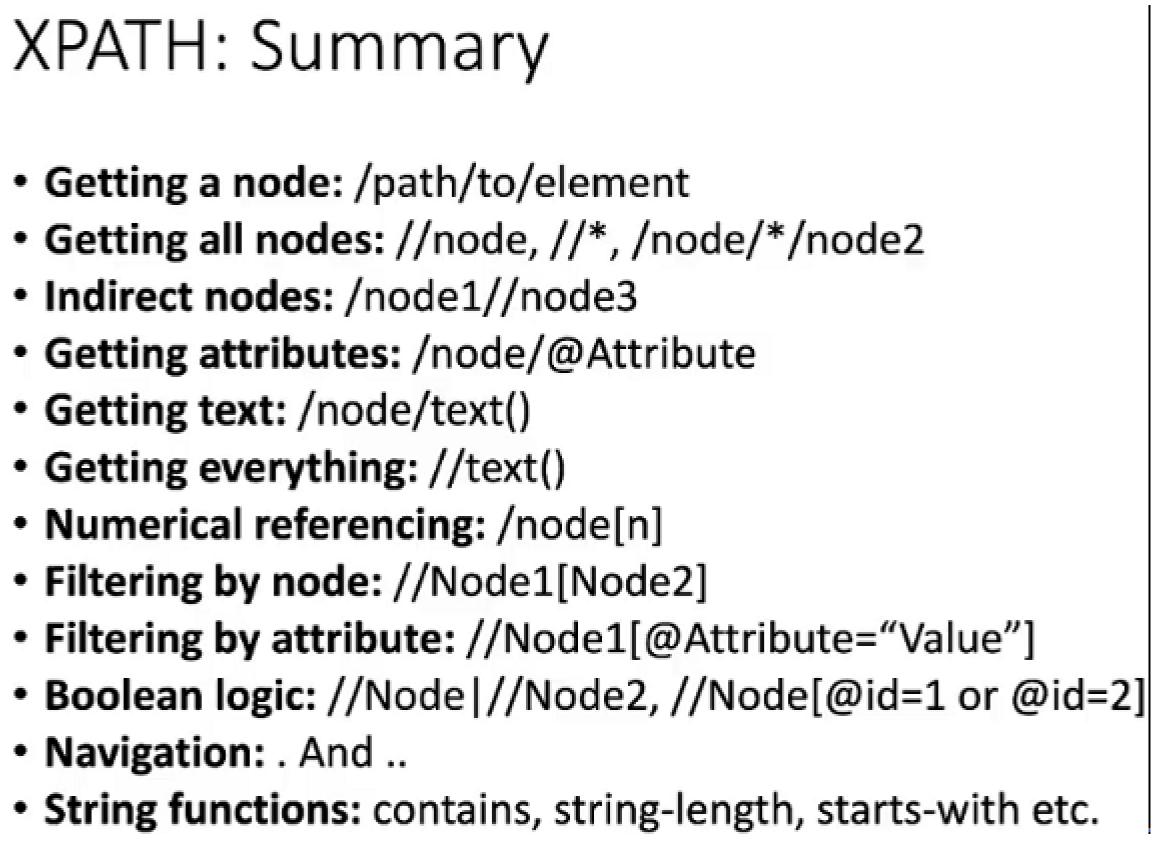
- There are times where data must be transferred across networks.
- But how can we efficiently pack information such as tables and objects and send it?
- There are three main formats to pick from:

XML

- Stands for eXtensible Markup Language
- Syntax is very similar to HTML
- Uses opening and closing tags
- You can represent any object with XML
- You can use XSLT to convert it to a readable format in HTML
- The difference between HTML and XML is that:
    - HTML describes presentation and
    - XML describes content
- XML is aimed toward being both human and machine readable
- Example:

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

XPath

- You'll also need to know a bit of XPath for the exam. XPath is a query language used for selecting XML nodes that fit specific conditions.
- I don't have enough time to write a tutorial of it here, but this site is pretty good for learning it: https://www.w3schools.com/xml/xpath_syntax.asp
- If you've used anything like jQuery before, XPath is pretty much the same, but with a slightly different syntax, which feels like a cross between Bash and jQuery.
- 

XPATH: Summary

- **Getting a node:** /path/to/element
- **Getting all nodes:** //node, //*, /node/*/node2
- **Indirect nodes:** /node1//node3
- **Getting attributes:** /node/@Attribute
- **Getting text:** /node/text()
- **Getting everything:** //text()
- **Numerical referencing:** /node[n]
- **Filtering by node:** //Node1[Node2]
- **Filtering by attribute:** //Node1[@Attribute="Value"]
- **Boolean logic:** //Node|//Node2, //Node[@id=1 or @id=2]
- **Navigation:** . And ..
- **String functions:** contains, string-length, starts-with etc.

JSON

- Stands for JavaScript Object Notation
- Used to represent objects in the form of arrays, key-value arrays and values.
- Uses curly braces
- More efficient than XML, but a little more limited
- Example:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

CSV

- Stands for comma-separated-values
- Commonly used for storing simple data values for processing
- Very minimalistic
- Example:

```
x,y
0,6.77
1,7.23
2,8.97
3,9.26
4,10.24
5,11.25
```

Remote access

- Remote access refers to a central system, with outside workstations running desktop environments remotely with it.
- Since there is a centralised computation, you can only have a single point of failure
- Efficient resource usage, meaning there is less to maintain
- It is very costly

Client / Server

- A client (the user) sends requests to a server for information.
- The client can be a computer, a phone, a thermostat, a TV, anything that the user is using to communicate with a server
- The server is a system that sends responses to requests. It can have a database, a file store etc.
- Using this, it is cheaper to create networks of small PCs, and it's scalable.
- However, you need to manage the server, and setting one up / managing one has a steep learning curve.

Thin clients

- A **thin client** is where the client only shows information from the server; it is optimised for the server doing most of the work.
- More specifically, it only implements the presentation layer
- Thin clients are easy to port to multiple platforms, but the server has a heavier workload
- Example: a browser on www.time.is, looking at the time

Fat clients

- A **fat client** is where the client allows you to interact with the server.
- More specifically, it implements the presentation and application layers
- Servers that use fat clients have less workload, but the clients are pinned down to their respective platforms
- Example: a desktop PC running administration software

Relational model

The relational data model

- There are two parts to a relational database:
    - Database management system (DBMS), software that manages database
    - Database (the actual data itself)

- There are also multiple kinds of database technologies, such as:
    - Relational: MySQL, SQLite, PostgreSQL
    - Non-relational (NoSQL): MongoDB, Couchbase

Independence

- Data independence: insulates how data is structured and stored from applications and users
- Logical independence: protection from changes in logical structure (schema/attributes)
- Physical independence: protection from changes in physical structure (how it is stored)

DBMS

- A DBMS is a set of computer programs that:
    - support a data model to represent the database
    - stores and manages large amounts of data
    - manages transactions and concurrency
    - controls access
    - is resilient (can recover from crashes)

Data model

- Data models are mathematical concepts for describing data
    - Relational model is the most widely used data model

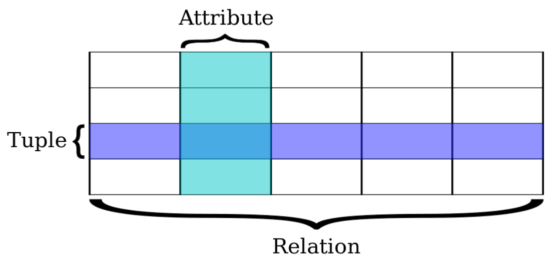
Data sublanguages

- Data languages have two parts:
    - data definition language (they define templates)
    - data manipulation language (CRUD: creation, reading, updating, deleting)

Relational data model

- A relational data model that uses relations.

- Formal: relation = subset of cartesian product of sets
- Informal: relations are like tables, or the set of records in a database table
- Each column is an 'attribute'
- A 'k-ary relation' is a relation with 'k' columns.
- Order of rows do not matter
- Used by Microsoft Access, SQL etc.



- A schema is a "blueprint" or a "template" for some k-ary relation.
    - Schemas store relation names and ordered sequences
    - The DBMS implements the schema to give definitions of types (e.g. name must be a string)
    - The database holds instances of the schema. The instances must follow the schema's properties.

Keys and functional dependencies

Key

- A **key** is an attribute, or a set of attributes, that uniquely identifies a record.
- For example, this could be a number (ID) or a first name + surname.

Functional dependency

- A **functional dependency** is a function that maps columns to other columns.
- An instance of a schema (basically a table with data in it) 'satisfies' a functional dependency if there are no instances where the same column values map to different column values.
- Example:

First name	**Last name**	**Age**	**Sex**
Matthew	Barnes	19	M
Elon	Musk	46	M
Melinda	Gates	53	F
hPdmrLZk	Barnes	99	F

- This table satisfies the functional dependency:

    - (first name, last name) → (age, sex)
- Because for each first name + last name, there is a unique age + sex.
- However, the table does not satisfy the functional dependencies:
    - (last name) → (age)
    - (last name) → (sex)
- Because when you input ‘Barnes’, you can get either ages ‘19’ or ‘99’ and either genders M or F.

- We can say that (A1, A2, A3...) **functionally determines** (B1, B2, B3...)
- (A1, A2, A3...) is the determinant and (B1, B2, B3...) is the dependent set.

Splitting / Combining rule

- If you have (A1, A2, A3...) → (B1, B2, B3...) then you can do:
    - (A1, A2, A3...) → B1
    - (A1, A2, A3...) → B2
    - (A1, A2, A3...) → B3 etc.

Trivial dependencies

- You can make columns functionally dependent on themselves:
    - (A1, A2, A3) → (A1, A2)
    - (A1, A2, A3) → A3
    - (A1, A2, A3) → (A1, A3) etc.
- More formally, you can do A → B where A is a set of columns and B is a subset of A.

Implication

- So, we know that a functional dependency is a function.
- Let’s just say we have a set of those functional dependencies, called ‘S’.
- Now let’s say we have another functional dependency called A → B.
- Lastly, let’s say that:
    - if all of the functional dependencies in ‘S’ are satisfied,
    - then A → B must be satisfied.
- You can write this like: S ⊧ A → B
- Example:
    - {A → B, B → C} ⊧ A → C
    - This is because of the rule of transitivity. If A → B is true, and B → C is true, then A → C must also be true.
- You can think of this like propositional logic, with ‘S’ being the set of formulas and A → B being a formula.

Equivalence

- Let’s say we have two sets of functional dependencies, S and T.
- S is equivalent to T if S ⊧ T and T ⊧ S.
- This means that S and T are equivalent if S is satisfied by the same functional dependencies as T.

A small problem

- We can only find functional dependencies that do NOT work with our schema instances, but not functional dependencies that will always work with it, except for trivial dependencies, like A → A.
- This is because we could always add a new entry that breaks our functional dependencies.
- Example:

First name	Last name	Age	Sex
Matthew	Barnes	19	M
Elon	Musk	46	M
Melinda	Gates	53	F
hPdmrLZk	Barnes	99	F

- At first the functional dependency (first name) → (age) is satisfied by this table. However...

First name	Last name	Age	Sex
Matthew	Barnes	19	M
Elon	Musk	46	M
Melinda	Gates	53	F
hPdmrLZk	Barnes	99	F
Matthew	Patrick	31	M

- Now, (first name) → (age) doesn't work, because when you input 'Matthew', you can get either '31' or '19'.
- Therefore, we can only get functional dependencies that do **not** hold for the schema, but not functional dependencies that will hold for the schema.

More keys

Superkey

- A superkey is a set of columns that can be a functional dependency of any other column in the schema.
- It can also be used to uniquely identify each record in the table.
- A schema can have multiple superkeys, all of different sizes.
- An ID is a perfect example of a superkey.

- In the table above, (first name, last name) is an example of a superkey.
- All of the columns in a schema will always be a superkey, e.g. (first name, last name, age, sex) is a superkey.

Candidate key

- A candidate key can be thought of as a special kind of superkey.
- It has the same properties of a superkey in that it can be a functional dependency of any other column in the schema.
- However, there is no subset of the candidate key that is also a superkey; the candidate key is the "smallest form" of superkey possible in the schema.
- In the table above, (age) would technically be a candidate key, because each record has a unique age. However, as soon as someone with the same age as someone else is added, (age) would stop being a candidate key.

Closure

- How do we find all candidate keys?
- To answer that, we need to find all the superkeys.
- To answer that, we need to figure out how to prove $F \models X \rightarrow Y$ (F is a set of functional dependencies)

- $X^+ = \{ A_i : F \models X \rightarrow A_i \}$ is the closure of X with respect to F
- So basically X^+ is a set of all the columns in the schema where, if the functional dependencies in F are true, then X → the columns in X^+ are true.

- $F \models X \rightarrow Y$ if and only if $Y \subseteq X^+$
- This basically means that the condition of "If everything in F is satisfied, X → Y is satisfied" is true if and only if Y has columns where X → the columns that are satisfied if F is satisfied.

- That sounds backwards, overly-complicated and unnecessary, but it allows us to computationally find superkeys and, in turn, candidate keys.
- If we don't find all of the columns in X^+, then we can safely assume that X is not a superkey. If it does have all the columns, then it is a superkey.

Closure algorithm

- The closure algorithm allows us to check if we have a superkey given:
    - a schema: R
    - a set of functional dependencies: F
    - a set of columns from R to check if it is a superkey: X
- This is how the closure algorithm goes:
    1. You start with the set X. You make that X_0.
    2. Look along F, and check if there are any Y → Z where Y is a subset of X.
    3. If there is, then union Z with X_0 and make that X_1. Now, we use X_1 instead of X_0.
    4. Keep repeating and looping around F until X_n just remains the same.

5. You should now have X^+.

- Here is an example:
  - R(A,B,C,D,E,H,G) - This is our schema
  - F = { AB → CD, C → EH, AC → H } - This is the set of functional dependencies
  - $\{A, C\}^+$ - This is what we want to find out

X_n	Value	Explanation
X_0	{A, C}	In the beginning, we start with the values in X, that is, the set we want to determine if it is a superkey or not.
X_1	{A, C} ∪ {E, H}	In F, there is a C → EH, and we have a ‘C’ in X_0. Therefore, we can union the result of that functional dependency with our X and increment its index.
X_1	{A, C, E, H}	It should now look like this.
X_2	{A, C, E, H} ∪ {H}	In F, there is an AC → H, and we have an ‘A’ and a ‘C’ in X_1, so we can union ‘H’ with our X.
X_2	{A, C, E, H}	Hmm, that didn’t seem to do anything. Additionally, there are no more functional dependencies of interest for us to look at. Therefore, this will be our output: this is $\{A, C\}^+$.

- Since $\{A, C\}^+$ has only {A, C, E, H} and not all of the columns in R, it’s not a superkey.
- Let’s try another example:
  - R(A,B,C,D,E,H,G) - This is our schema
  - F = { AB → CD, C → EH, D → G } - This is the set of functional dependencies
  - $\{A, B\}^+$ - This is what we want to find out

X_n	Value	Explanation
X_0	{A, B}	In the beginning, we start with the values in X, that is, the set we want to determine if it is a superkey or not.
X_1	{A, B} ∪ {C, D}	In F, there is an AB → CD, and we have an ‘A’ and a ‘B’ in X_0. Therefore, we can union the result of that functional dependency with our X and increment its index.
X_1	{A, B, C, D}	It should now look like this.
X_2	{A, B, C, D} ∪ {E, H}	In F, there is a C → EH, and we now have a ‘C’ in X_1 so we can union ‘H’ and ‘E’ with our X.
X_2	{A, B, C, D, E, H}	Now we have this.
X_3	{A, B, C, D, E, H} ∪ {G}	In F, there is a D → G. Since we have ‘D’ in X_3, we can union ‘G’ with our X.
X_3	{A, B, C, D, E, H, G}	We’re done here now; there are no more functional dependencies of interest. This is $\{A, B\}^+$.

- {A, B}$^+$ has all of the elements of R in it, implying that {A, B} is a superkey.
- {A, B} is also a candidate key, because {A}$^+$ is {A} and {B}$^+$ is {B}; it is the smallest form of superkey with this schema.

Bad relations and anomalies

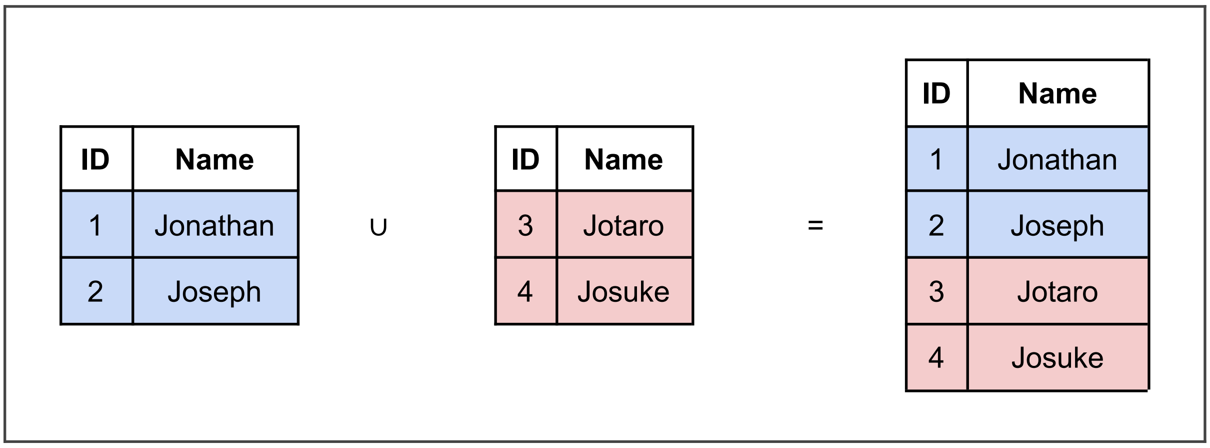
- By going through this process, we can check for bad relations.
- For example, X → A is good if X is a superkey.
- But if X is not a superkey, that's a sign of a bad relation.

- By leaving bad relations unchanged, they can cause anomalies:
  - Redundancy: information unnecessarily repeated
  - Update Anomalies: change information in one tuple and leave same info unchanged in another
  - Insert Anomalies: we could insert data incorrectly
  - Deletion anomalies (e.g., leaving a field as empty)

Relational algebra

Union

- When you union two relations, the elements of one are added to the other's table.
- Basically, you're adding records to a table when you 'union'.
- Union only works when both relations have the <u>same type of columns and number of columns</u>; it cannot work otherwise!
- 'Union' is represented with the ∪symbol.
- Example:



ID	Name
1	Jonathan
2	Joseph

∪

ID	Name
3	Jotaro
4	Josuke

=

ID	Name
1	Jonathan
2	Joseph
3	Jotaro
4	Josuke

Difference

- When you perform the 'difference' operation on two relations, you're 'taking away' elements from the left-most table.
- This operation returns the left table, but without elements from the right table.

- ‘Difference’ is represented with the minus (-) symbol.
- Example:

ID	Name
1	Jonathan
2	Joseph
3	Jotaro
4	Josuke

$-$

ID	Name
2	Joseph
4	Josuke

$=$

ID	Name
1	Jonathan
3	Jotaro

Cartesian product

- The cartesian product works in the same way as in set theory.
- This will return a table with every possible pairing of records in the left table and records in the right table.
- Example:

Joestar	
ID	**Name**
1	Jonathan
2	Joseph

$\times$

Fruit	
Name	**Colour**
Banana	Yellow
Apple	Red

$=$

Joestar x Fruit			
ID	**Joestar. Name**	**Fruit. Name**	**Colour**
1	Jonathan	Banana	Yellow
1	Jonathan	Apple	Red
2	Joseph	Banana	Yellow
2	Joseph	Apple	Red

Projection

- The projection operation works like a ‘filter’.
- It removes any unwanted columns from our table and only leaves the ones we want.
- It is represented with the pi symbol: π
- The syntax goes like this:
  - $\pi_{<attribute\ list>}(<\ relation\ name\ >)$
  - Where ‘<attribute list>’ is the list of columns that we want to keep, and ‘<relation name>’ is the name of the table to filter.

- Example:

Joestar x Fruit			
ID	Joestar. Name	Fruit. Name	Colour
1	Jonathan	Banana	Yellow
1	Jonathan	Apple	Red
2	Joseph	Banana	Yellow
2	Joseph	Apple	Red

$\pi_{Joestar.Name,\ Colour}$(Joestar x Fruit)	
Joestar. Name	Colour
Jonathan	Yellow
Jonathan	Red
Joseph	Yellow
Joseph	Red

Selection

- The 'selection' operator allows you to check the records alongside a condition.
- If the condition shows to be false, then don't include this record in the result.
- If the condition shows to be true, then include this record in the result.
- Its syntax goes like this:
    - $\sigma_{\Theta}(< relation\, name >)$
    - Where 'Θ' is the condition to use, and '<relation name>' is the name of the table of records to check.
- The condition can use the syntax of predicate logic.
- Example:

Joestar	
ID	Name
1	Jonathan
2	Joseph
3	Jotaro
4	Josuke

$\sigma_{ID>2}$(Joestar)	
ID	Name
3	Jotaro
4	Josuke

$\sigma_{ID=1 \wedge Name=Jonathan}$ (Joestar)	
ID	Name
1	Jonathan

- The last example is a bit unnecessary in practice, but it shows how you can use predicate logic syntax in the condition.

Renaming

- The renaming operator simply renames columns from one name to another.
- This is especially useful in the ‘union’ operation, as you can rename columns from one table to the same column names of another, and then ‘union’ them together.
- The syntax goes like this:
    - $\rho_{<oldname>\rightarrow<newname>}(< relation\, name >)$
    - With ‘<oldname>’ being the old name of the column, ‘<newname>’ being the new name of the column, and ‘<relation name>’ being the name of the relation whose columns we should rename.
    - You can also rename multiple columns by doing this:
    - $\rho_{O_1\rightarrow N_1, O_2\rightarrow N_2, O_3\rightarrow N_3}(< relation\, name >)$
    - Which would return the same table, but with the columns O_1, O_2 and O_3 renamed to N_1, N_2 and N_3 respectively.
- Example:

Joestar	
ID	**Name**
1	Jonathan
2	Joseph
3	Jotaro
4	Josuke

$\rho_{Name\rightarrow Surname}$**(Joestar)**	
ID	**Surname**
1	Jonathan
2	Joseph
3	Jotaro
4	Josuke

Θ-Join

- The Θ-Join isn’t really an operation that you use; it’s more of a “special query” of a certain form.
- It looks like this:
    - $\sigma_{\Theta}(R_1 \times R_2)$
    - Which is basically a selection operation, but with the cartesian product of two relations.
- This is typically used to join two relations together.
- Example:

Joestar	
ID	Name
1	Jonathan
2	Joseph
3	Jotaro
4	Josuke

Birthyears	
ID	Birthyear
1	1868
2	1920
3	1970
4	1983

$\sigma_{Joestar.ID=Birthyears.ID}$**(Joestar x Birthyears)**			
Joestar. ID	Birthyears. ID	Name	Birthyear
1	1	Jonathan	1868
2	2	Joseph	1920
3	3	Jotaro	1970
4	4	Josuke	1983

- This is a bit inefficient... we have repeated columns we don't want, like Joestar.ID and Birthyears.ID.
- This is easily fixed with the projection operation, because it gets rid of columns we don't need.
- Now our desired relation calculation looks like this:
    - $\pi_{Name,\, Birthyear}(\sigma_{Joestar.ID=Birthyears.ID}$**(Joestar x Birthyears)**$)$
- That's a bit of a mouthful, and we're probably going to be doing this kind of operation a lot. It would be so much easier if this was all packaged into one operation for us to use.

Natural join

- Natural join is the name of a special kind of operation that *joins* multiple relations together using Θ-Join, and also uses the projection operation to discard repeated columns.
- Its syntax goes like this:
    - $R_1 \bowtie R_2$
    - Which really means:
    - $\pi_{<list>}(\sigma_{R_1.A_1=R_2.A_1 \wedge R_1.A_2=R_2.A_2 \wedge ...}$**($R_1$ x R_2)**$)$
    - Where '<list>' is a list of all the columns in R_1 and R_2 (which are both tables) except the ones in the condition (A_1, A_2 etc.)
- This basically does what we were trying to do at the end of the subcategory 'Θ-Join', except it's packaged into one operation.
- Example:

Joestar	
ID	**Name**
1	Jonathan
2	Joseph
3	Jotaro
4	Josuke

Birthyears	
ID	**Birthyear**
1	1868
2	1920
3	1970
4	1983

Joestar ⋈ Birthyears		
ID	**Name**	**Birthyear**
1	Jonathan	1868
2	Joseph	1920
3	Jotaro	1970
4	Josuke	1983

Aggregate functions with unary operator

- You can use aggregate functions in relational algebra using the unary operator ‘γ’.
- The syntax goes like this:
    - $_{\text{[columns to group by]}}\gamma_{\text{[aggregate function calls]}}$([relation to do the aggregate functions on])
- So, for example, if you wanted to find the average of all student scores, you could do something like:
    - $_{\text{STUDENT\_ID}}\gamma_{\text{AVG(SCORE)} \rightarrow \text{AVERAGE\_SCORE}}$(STUDENT_RESULTS)
- The output only contains the aggregate functions’ columns and the group by columns, so the example’s output would be:

STUDENT_ID	AVERAGE_SCORE
1	5.78
2	4.89
3	3.68
4	8.45

Database systems

Normalisation

- Normalisation avoids redundant data, to make databases more efficient, keep its size smaller and makes it easier to maintain.
- There are 4 normal forms:
    - First normal form
    - Second normal form
    - Third normal form
    - Boyce-Codd normal form

Zeroth normal form

- A flat file database.
- Example:

Stand users				
Stand user	**Stand name**	**Birthyear**	**Part**	**Stand Type**
Joseph Joestar	Hermit Purple	1920	Part 2, Part 3, Part 4	Tool Stand
Joseph Joestar	Hermit Purple	1920	Part 2, Part 3, Part 4	Integrated Stand
Jotaro Kujo	Star Platinum	1970	Part 3, Part 4, Part 5, Part 6	Close-range Stand
Jotaro Kujo	Star Platinum	1970	Part 3, Part 4, Part 5, Part 6	Range Irrelevant
Josuke Higashikata	Crazy Diamond	1983	Part 4	Close-range Stand
Yoshikage Kira	Killer Queen	1966	Part 4	Close-range Stand

- This is in zeroth normal form because it does not comply with first normal form and is simply one big table.

First normal form

- To be in first normal form, each cell must have one piece of information in it, and there should be no duplicate tables.
- You should also be able to uniquely identify records using a primary key, made up of a number of columns.
- Example:

Stand users			
Stand user (PK)	**Stand name**	**Birthyear**	**Stand type (PK)**
Joseph Joestar	Hermit Purple	1920	Tool Stand
Joseph Joestar	Hermit Purple	1920	Integrated Stand
Jotaro Kujo	Star Platinum	1970	Close-range Stand
Jotaro Kujo	Star Platinum	1970	Range Irrelevant

Josuke Higashikata	Crazy Diamond	1983	Close-range Stand
Yoshikage Kira	Killer Queen	1966	Close-range Stand

Stand user parts	
Stand user (PK)	**Part (PK)**
Joseph Joestar	Part 2
Joseph Joestar	Part 3
Joseph Joestar	Part 4
Jotaro Kujo	Part 3
Jotaro Kujo	Part 4
Jotaro Kujo	Part 5
Jotaro Kujo	Part 6

- This is now in first normal form because repeated data (Part) has been split into its own table.
- Note that the table 'Stand user parts' has nothing but a primary key; this means that 'Stand user parts' does not have any non-prime attributes.

Second normal form

- To be in second normal form, all attributes must be functionally dependent with the **entire** primary key.
- More simply, for each attribute, the following must be true: $K \rightarrow A_i$
- Example:

Stand users		
Stand user (PK)	**Stand name**	**Birthyear**
Joseph Joestar	Hermit Purple	1920
Jotaro Kujo	Star Platinum	1970
Josuke Higashikata	Crazy Diamond	1983
Yoshikage Kira	Killer Queen	1966

Stand user parts	
Stand user (PK)	**Part (PK)**

Joseph Joestar	Part 2
Joseph Joestar	Part 3
Joseph Joestar	Part 4
Jotaro Kujo	Part 3
Jotaro Kujo	Part 4
Jotaro Kujo	Part 5
Jotaro Kujo	Part 6

Stand types	
Stand name (PK)	**Stand type (PK)**
Hermit Purple	Tool Stand
Hermit Purple	Integrated Stand
Star Platinum	Close-range Stand
Star Platinum	Range Irrelevant
Crazy Diamond	Close-range Stand
Killer Queen	Close-range Stand

- This is now in second normal form. In the ‘Stand users’ table before, the ‘Birthyear’ column was functionally dependent on ‘Stand user’, but had nothing to do with ‘Stand type’. This meant that ‘Stand users’ was not functionally dependent on the whole primary key, meaning it was not in 2NF.
- To fix that, ‘Stand type’ was moved to its own table. Now, the ‘Stand users’ table only has a primary key of one column: ‘Stand user’. Every column of ‘Stand users’ is functionally dependent on the primary key, making it 2NF.

Third normal form

- For each $K \rightarrow A_i$, there should be no $K \rightarrow A_i \rightarrow A_j$
- Basically, every attribute should **only** be dependent on the primary key.
- To check if your table is in 3NF, try going from the primary key to a non-prime attribute, and then from that attribute to another non-prime attribute, all via functional dependencies. If you can do that, then your table is not in 3NF.
- An example of a table in 2NF but not in 3NF is:

Protagonists

Part (PK)	Protagonist	Stand name
Part 2	Joseph Joestar	Hermit Purple
Part 3	Jotaro Kujo	Star Platinum
Part 4	Josuke Higashikata	Crazy Diamond

- If you know ‘Part’, you can work out ‘Protagonist’ and ‘Stand name’.
- However, if you have ‘Protagonist’, you can also work out ‘Stand name’. That means that “Part → Protagonist → Stand name” exists, meaning this table is not in 3NF.
- To fix this, we could create a separate table:

Protagonists	
Part (PK)	**Protagonist**
Part 2	Joseph Joestar
Part 3	Jotaro Kujo
Part 4	Josuke Higashikata

Stands	
Protagonist (PK)	**Stand name**
Joseph Joestar	Hermit Purple
Jotaro Kujo	Star Platinum
Josuke Higashikata	Crazy Diamond

- Now, the table is in 3NF.

Boyce-Codd normal form

- Every functional dependency you have, e.g. A → B, one of the following must be true:
    - ‘A’ must be a superkey
    - The dependency is a trivial functional dependency (B ⊆ A)
- To check if your table is in BCNF, find a functional dependency A → B where ‘A’ is not a superkey. If you can find one, then your table is not in BCNF.
- An example of a table in 3NF and not in BCNF is the following:

Enrolment		
Student ID (PK)	**Subject (PK)**	**Professor**

101	Java	P.Java
101	C++	P.Cpp
102	Java	P.Java2
103	C#	P.Csharp
104	Java	P.Java

- The primary key is Student ID and Subject.
- However, there is a dependency between Subject and Professor:
    - (Professor) → (Subject)
- While Subject is a prime attribute, Professor itself is a non-prime attribute (not a superkey) which is not allowed by BCNF.
- To fix this, we decompose it into 2 tables:

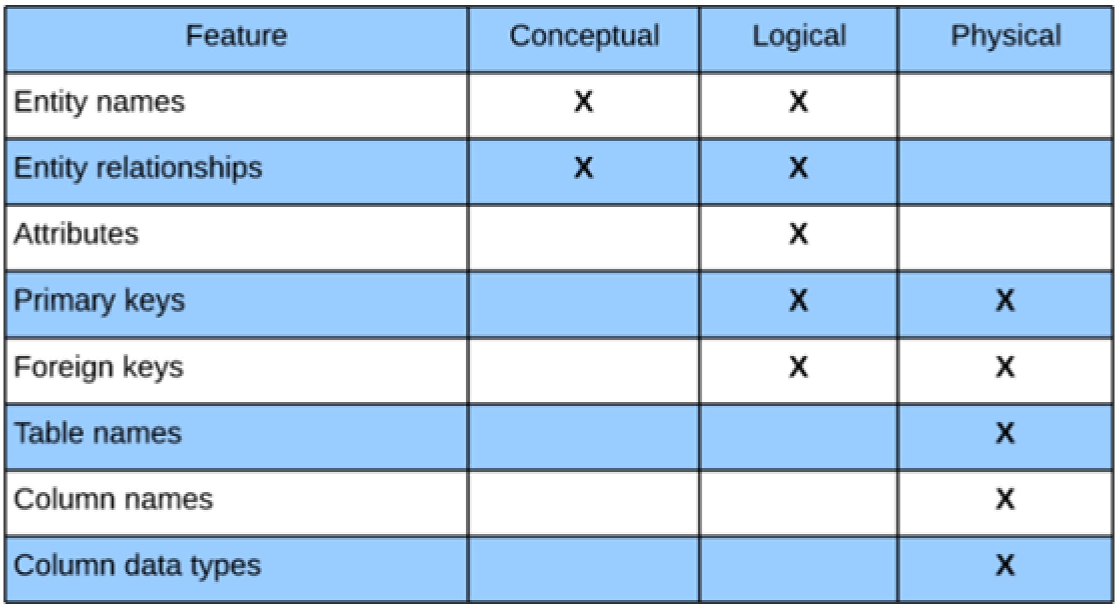
Students-Professors	
Student ID (PK)	**Professor (PK, FK)**
101	P.Java
101	P.Cpp
102	P.Java2
103	P.Csharp
104	P.Java

Professors-Subjects	
Professor (PK)	**Subjects**
P.Java	Java
P.Cpp	C++
P.Java2	Java
P.Csharp	C#

- In the Students-Professors table, all functional dependencies are trivial (they all relate to the primary key itself).
- In the Professors-Subjects table, the only functional dependency that exists is (Professor) → (Subjects), and (Professor) is a superkey, so BCNF is present within both tables.

Modelling and SQL basics

Types of data modelling



Feature	Conceptual	Logical	Physical
Entity names	X	X	
Entity relationships	X	X	
Attributes		X	
Primary keys		X	X
Foreign keys		X	X
Table names			X
Column names			X
Column data types			X

Conceptual (ideas)

- Include important entities and the relationship between them.
- Do not specify attributes.
- Do not specify primary keys.
- Used as the foundation for logical data models.
- Think of it as a very loose class diagram, with no operations or attributes.

Logical (high level)

- Include all entities and relationships between them.
- Specify attributes for each entity.
- Specify primary key for each entity.
- Specify foreign keys, which identify the relationship between different entities.
- Involve normalization
- Do NOT use indexes here, because this will harm your normalisation efforts.

Physical (low level)

- Specify all tables and columns.
- Include foreign keys to identify relationships between tables.
- May include normalization, depending on user requirements.
- May be significantly different from the logical data model.
- Will differ depending on which DBMS (database management system) is used.
- You can (and should) use indexes where possible here.

Entities

- An entity is an object in our system, with attributes and relations to other entities.
- There are different types of entities:

Strong

- Strong entities exist independently from other entities. They always have a key, and can exist on their own.

Weak

- Weak entities need to depend on a strong entity to exist. This includes things like types (for example, Colour would be weak and dependent on Shape, which would be strong).

Associative

- Associative entities describe the instances of entities as opposed to the actual entity template itself, like join tables (which allow many-to-many relationships). They're usually empty at the start, and fill up to describe the entity instances.

Attributes

Attributes

- A characteristic of an entity or a relationship, e.g. a student's score.

Multivalued attributes

- An attribute that contains more than one value, e.g. a full name could be made up of a 'first name' and a 'last name'.

Derived attributes

- An attribute that can be calculated from other attributes, e.g. a score average.

Relationships

Relationship

- A relationship is a connection between two entities. For example, a Student can have a relationship between a Lecturer.

Weak relationship

- A weak relationship is the relationship between a weak relation to its owner (like from Colour to Shape).

Cardinality

- An entity can have a various quantity of another entity. The relationship describes this using a cardinality.
- The cardinality refers to the maximum number of entities one can have.
- The modality refers to the minimum number of entities one can have.

One-to-one

- Each entity only has one of each other, for example a Shape can only have one Colour.

One-to-many

- One entity has lots of the other entity, but that other entity can only belong to one of the other entity. For example, a Painter can have lots of Paintbrushes, but a Paintbrush can only belong to one Painter.

Many-to-many

- One entity can have lots of the other entity, and the other entity can have lots of the other entity as well. For example, a Student can have lots of Classes and a Class can have lots of Students.

SQL - Structured Query Language

- SQL is a query language that allows you to store and manipulate data.

Data definition language (DDL)

- Tables and views (virtual tables)
- Convert a data model to a (physical) database
- Actually stores the data

Data manipulation language (DML)

- Manipulates the data programmatically
- Declarative (where you say what you want, not how you do it)
- INSERT, DELETE, UPDATE or retrieve (SELECT) data.

Data definition

Create database

- You can create a database by using the ‘CREATE DATABASE’ operation:

```
CREATE DATABASE databasename;
```

Create table

- To create a table, you can use the ‘CREATE TABLE’ operation:

```
DROP TABLE IF EXISTS Student;
CREATE TABLE Student
(
  ID integer,
  email varchar(20),
  lastName varchar(20),
  firstName varchar(20),
  DOB date
);
```

Define primary keys

- You can define primary keys in ‘CREATE TABLE’ by specifying that a column is a primary key with the ‘PRIMARY KEY’ keywords:

```
CREATE TABLE Student
(
  ID integer PRIMARY KEY,
  email varchar(20) UNIQUE NOT NULL,
  lastName varchar(20),
  firstName varchar(20),
  DOB date
);
```

Define foreign keys

- To define a foreign key, you use the ‘FOREIGN KEY’, followed by the column to make the foreign key, followed by ‘REFERENCES’ followed by the column it references in another table, using the syntax *table_name*(*column*):

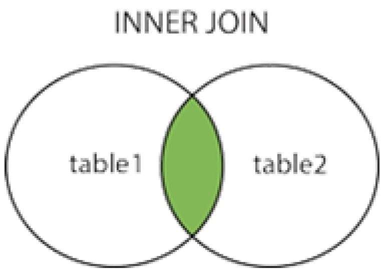
```
CREATE TABLE Student
(
  ID integer PRIMARY KEY,
  class integer,
  email varchar(20) UNIQUE NOT NULL,
  lastName varchar(20),
  firstName varchar(20),
  DOB date,
  FOREIGN KEY (class) REFERENCES Class(ID)
);
```

Joins

- To join means to attach multiple tables together, usually through use of foreign keys.
- There are different kinds of joins:

Inner join

- An inner join returns a table where both entries from table 1 and table 2 have foreign keys for each other; no 'null' values in the foreign key column:



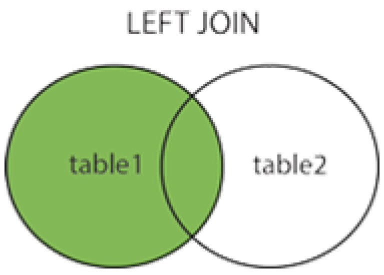
Characters	
Character	Stand ID
Jotaro Kujo	1
Polnareff	2
Jonathan Joestar	null

Stands	
Stand ID	Stand
1	Star Platinum
2	Silver Chariot
3	The World

Inner Join		
Character	Stand ID	Stand
Jotaro Kujo	1	Star Platinum
Polnareff	2	Silver Chariot

Left join

- A left join returns a table containing all the records from table 1, along with properties from table 2, if it exists. If it does not exist, then the table 2 columns will be null.



Characters	
Character	Stand ID
Jotaro Kujo	1
Polnareff	2

Stands	
Stand ID	Stand
1	Star Platinum

Left Join		
Character	Stand ID	Stand
Jotaro Kujo	1	Star Platinum

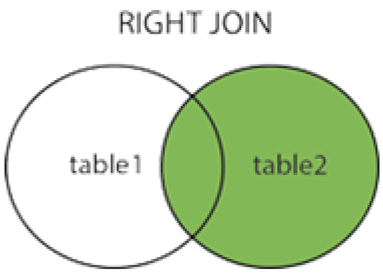
Jonathan Joestar	null

2	Silver Chariot
3	The World

Polnareff	2	Silver Chariot
Jonathan Joestar	null	null

Right join

- A right join is like a left join, but it contains all the records from table 2, along with properties from table 1, if it exists, and if it doesn't, they'll be 'null':



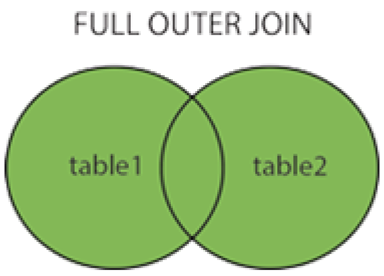
Characters	
Character	Stand ID
Jotaro Kujo	1
Polnareff	2
Jonathan Joestar	null

Stands	
Stand ID	Stand
1	Star Platinum
2	Silver Chariot
3	The World

Right Join		
Character	Stand ID	Stand
Jotaro Kujo	1	Star Platinum
Polnareff	2	Silver Chariot
null	3	The World

Full outer join

- This will return all records from table 1 and table 2, and if any foreign keys are null, then the respective columns will hold 'null'.



Characters	
Character	Stand ID
Jotaro Kujo	1
Polnareff	2
Jonathan Joestar	null

Stands	
Stand ID	Stand
1	Star Platinum
2	Silver Chariot
3	The World

Full Outer Join		
Character	Stand ID	Stand
Jotaro Kujo	1	Star Platinum
Polnareff	2	Silver Chariot
Jonathan Joestar	null	null
null	3	The World

Data languages

Relational Algebra vs SQL

Counterparts

SQL	**Relational Algebra**
SELECT	Projection π
FROM	Cartesian product $\times$
WHERE	Selection σ_Θ

Self-joins

- Let's say you had
- FATHER(father-name, child-name)

- How would you get
- GRANDFATHER(grandfather-name, grandchild-name)

- You would have to do a self-join, which is a TABLE × TABLE (a cartesian product on two equal relations)
- In this example, you could do FATHER × FATHER

Aliases in SQL

- In SQL, you can give things different names to make sense of things easier.
- You use the 'AS' keyword to give a column an alias.
- For example, if you did a self-join and did:
  - SELECT * FROM TABLE, TABLE
- You could differentiate the two tables using aliases, like this:
  - SELECT * FROM TABLE AS T1, TABLE AS T2

Multisets

- A multiset is like a set, but you can have duplicate values.
- For example, {4, 6, 7} can be a set, but {4, 4, 6, 6, 7, 7} is a multiset.

- You use the **μ(x,B)** operation to find the number of occurrences in a multiset, with '**x**' being the element and '**B**' being the multiset.
- For example, μ(5, {4, 5, 5, 6}) would be 2 because there are 2 '5's in the multiset.

Union

- A union with multisets is the same as sets, but you append all the elements together:
- {1, 5, 7} ∪ {5, 7, 9} = {1, 5, 5, 7, 7, 9}

- $\mu(t, R \cup S) = \mu(t, R) + \mu(t, S)$
- The occurrence of an element in the union is equal to the occurrences in one set plus the occurrences in the other set.

Difference

- Difference with multisets is the same as sets, but quantities matter.
- {1, 3, 3, 4, 5, 6} - {1, 3, 4, 6} = {3, 5}

- $\mu(t, R - S) = \max\{\mu(t, R) - \mu(t, S), 0\}$
- The occurrence of an element in the difference is the occurrences in the first set minus the occurrences in the second set. If that number is negative, then it's 0.

Intersection

- Intersection with multisets is the same as sets, but again, quantities matter.
- {4, 5, 5, 6} ∩ {5, 5, 6, 7} = {5, 5, 6}

- $\mu(t, R \cap S) = \min\{\mu(t, R), \mu(t, S)\}$
- The occurrence of an element in the intersection is the occurrences in the first set, or the occurrences in the second set; whichever one has the smallest number of occurrences.

Cartesian product

- Cartesian product with multisets is the same as for sets, but you can have duplicate pairings.
- {1, 1, 4, 7} ✕ {1, 4, 5, 7} = { {1, 1}, {1, 4}, {1, 5}, {1, 7}, {1, 1}, {1, 4}, {1, 5}, {1, 7}, {4, 1}, {4, 4}, {4, 5}, {4, 7}, {7, 1}, {7, 4}, {7, 5}, {7, 7} }

- $\mu(tt', R \times S) = \mu(t, R)\ \mu(t', S)$
- The occurrence of a pairing in the cartesian product is the occurrence of the first element in the first set times the occurrence of the second element in the second set.

Multisets in SQL

- Relational algebra automatically removes duplicates, but SQL may have multisets in them if not explicitly set otherwise.
- It's kept in SQL because it serves a functionality (e.g. it could be used for averages)
- To get rid of duplicates, use the 'DISTINCT' keyword after 'SELECT'.

Advanced SQL

SQL Aggregate functions

- avg(x) - Calculate the average of all the values in a column
- count(*) - Count number of entries in a table
- count(X) - Count number on non-null values in a column
- max(X) - Return the highest value in a column
- min(X) - Return the smallest value in a column
- sum(X) - Returns the sum of all the values in a column

Grouping

- You can group together aggregate functions using 'GROUP BY'
- So, for example, if you had a 'Students' and 'Scores' columns, and you want the average of everyone's scores, you'd want to group by 'Students', because they're the subject of your operation.

Views

- A view is like a virtual table
- A view doesn't store data, but it formats and collects data from other tables and displays it.
- It's kind of like a query shortcut.
- You can define it like this:
- CREATE VIEW ViewName AS [relational algebra or equivalent goes here]
- It's hard to update values using a view, because it's ambiguous as to what should change.

Indexes

- Indexes are numbered columns that uniquely identify a record.
- They're quick and easy to use, searching can be done in logarithmic time.
- They're usually implemented using binary trees, or hash indexes.
- You can create indexes by using the 'CREATE INDEX' operation.
- To create unique indexes, use 'CREATE UNIQUE INDEX'.

Data Protection and the GDPR

Data Protection

- Everything is data: social media, bank accounts, research etc.
- **Personal data**: information about an identifiable natural person.
- **Processing**: any sort of action applied upon personal data or sets of personal data.
- **Data controller**: one who controls the processing of personal data.
- **Data processor**: one who processes personal data on behalf of the controller.

GDPR

Key principles

- Lawful basis for processing
    - Explicit consent or necessity
- Pseudonymisation
    - Transforming personal data so that it cannot be attributed to you
- Right of access
- Right to erasure / right to be forgotten
- Data protection by design and by default
- Records of processing activities

Consent

- The biggest part of GDPR is consent.
- Essentially, you must agree for your data to be processed
- Already the case in the UK, but the requirements are becoming stronger

- No longer acceptable:
    - I accept the terms and conditions [which happen to be 20 pages long and include vaguely-worded consent for data processing]
    - Calls are recorded for training purposes
    - Opt-out systems
    - Pre-ticked consent checkboxes