

What we're discussing

Server

1. Cluster module and horizontal scaling ✓
2. Capacity Estimation, ASGs and Vertical scaling ✓
3. Load balancers ✓

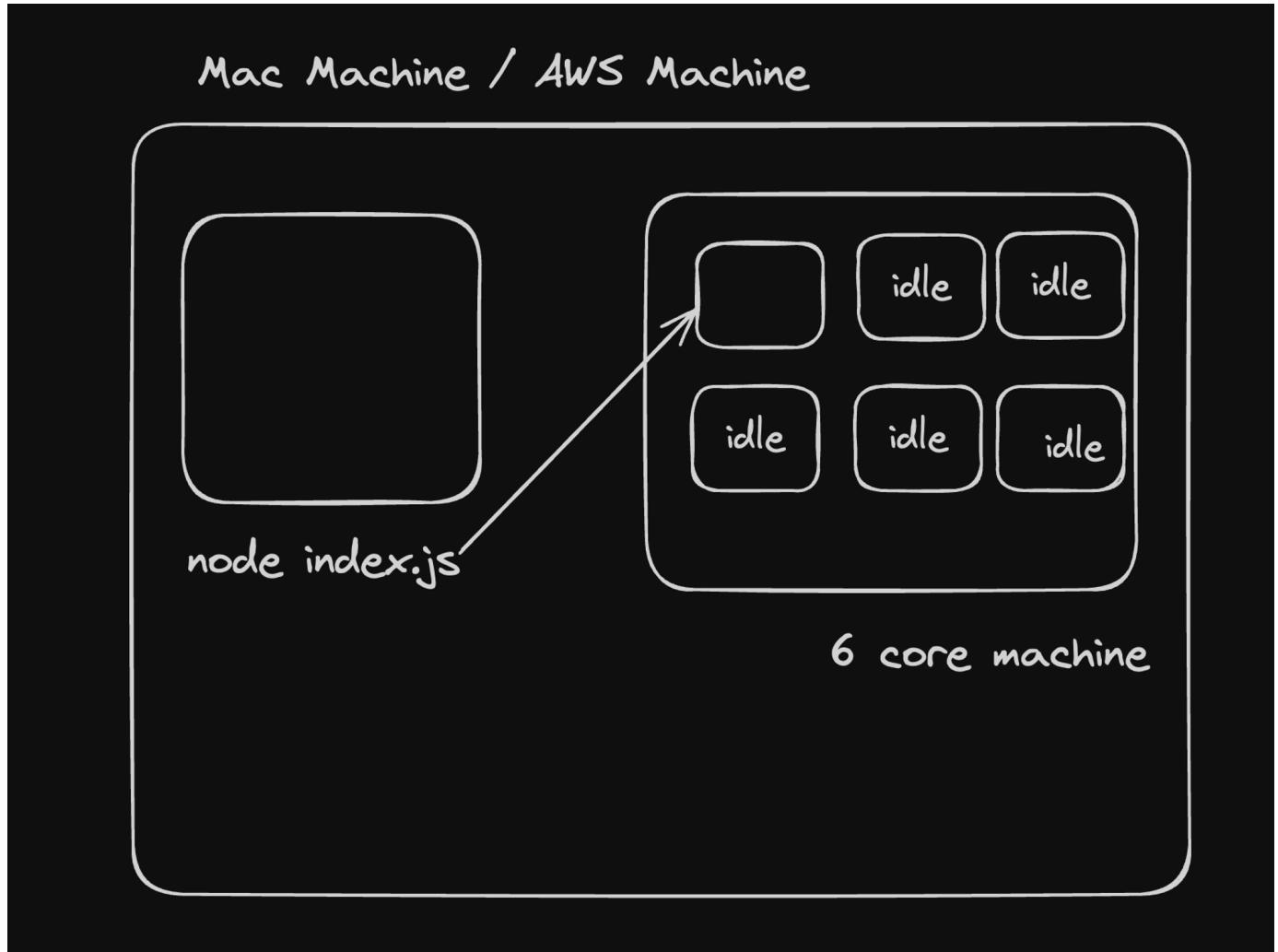
Database

1. Indexing ✓
2. Normalization
3. Sharding

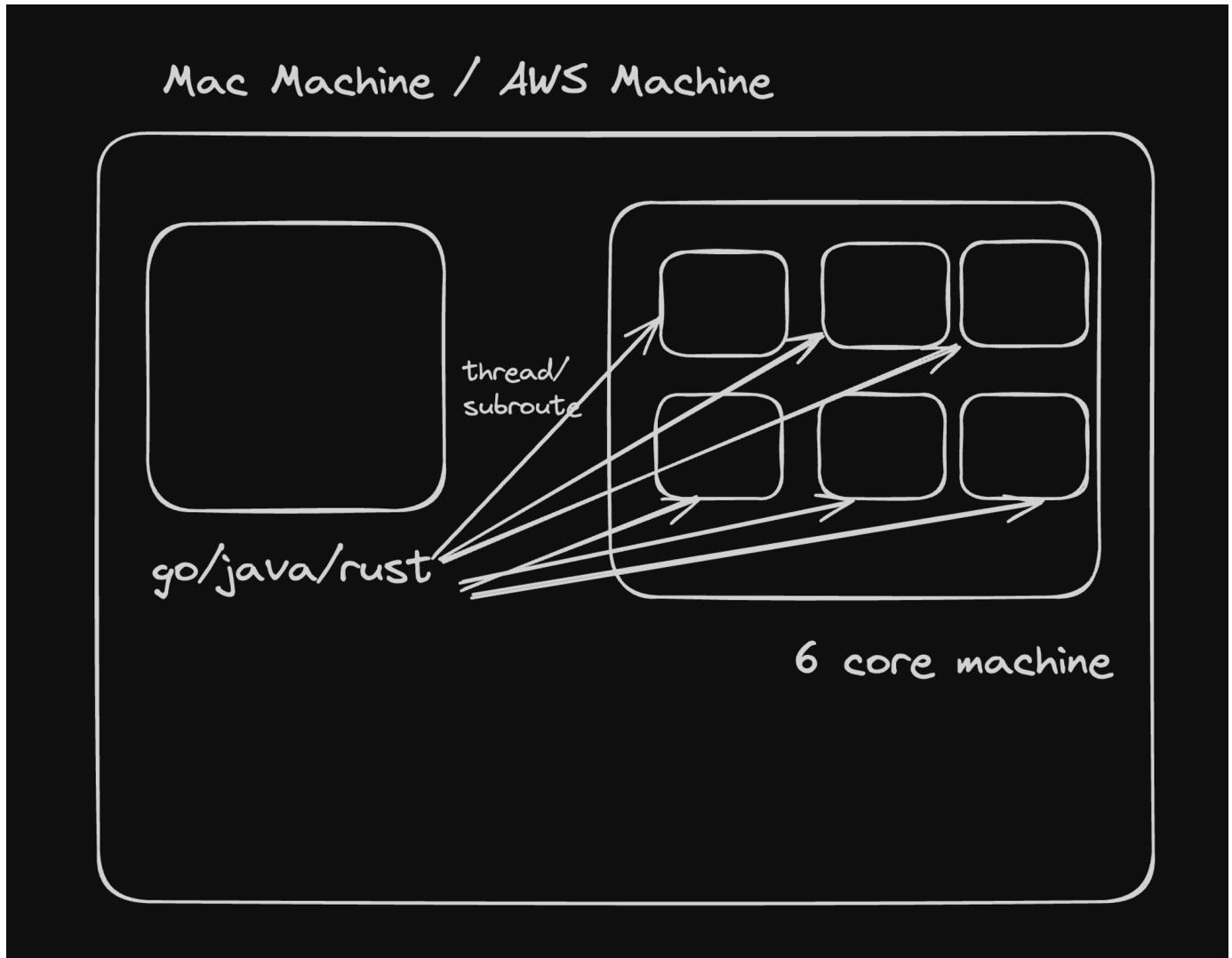
Vertical scaling

Vertical scaling means increasing the size of your machine to support more load

Single threaded languages



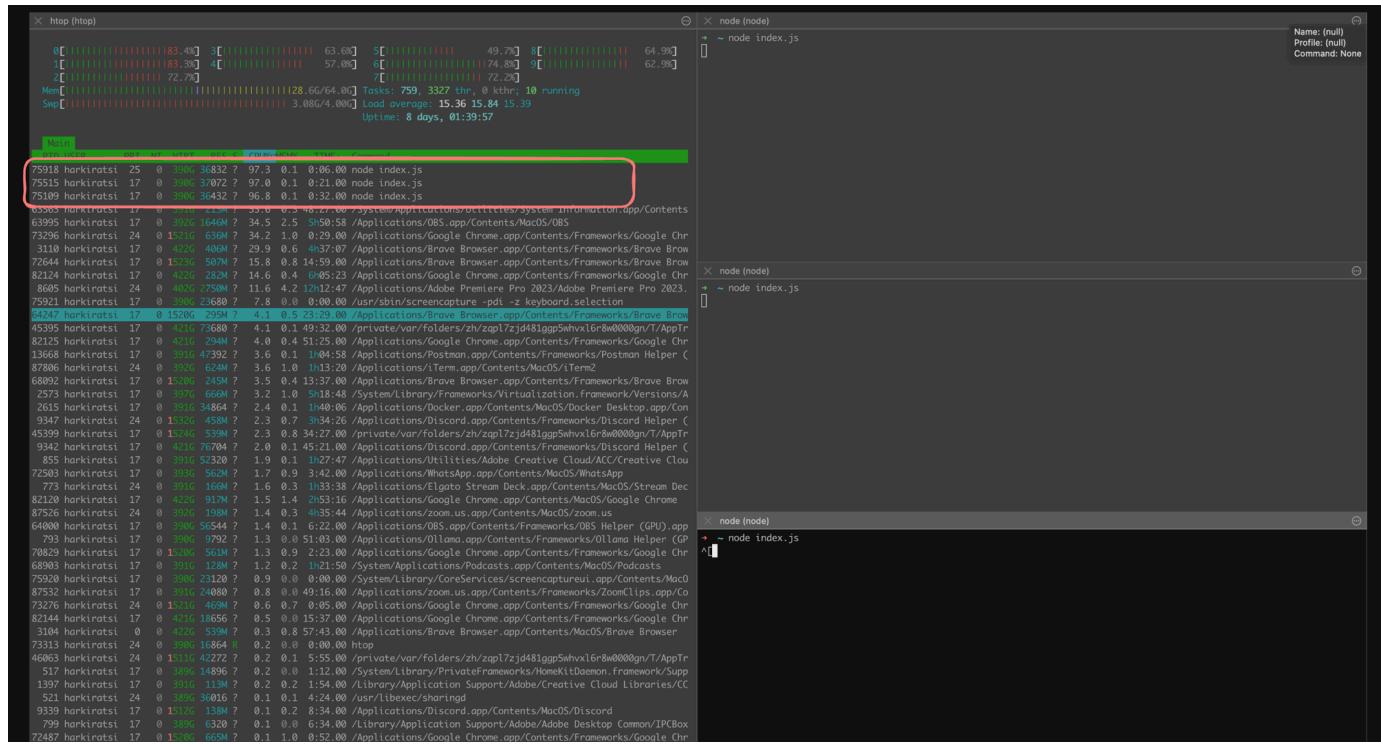
Multi threaded languages



Node.js

Let's run an infinite loop in a JS project and see how our CPU is used

```
let c = 0,  
while (1) {  
    c++;  
}
```



This confirms that only a single core of the machine is being used. We got 3 different processes using 100% CPU each.

Rust

```
use std::thread;

fn main() {
    // Spawn three threads
    for _ in 0..3 {
        thread::spawn(|| {
            let mut counter: f64 = 0.00;
            loop {
                counter += 0.001;
            }
        });
    }

    loop {
        // Main thread does nothing but keep the program alive
    }
}
```

Implementing horizontal scaling in Node.js project

You can start multiple node projects then? If there are 8 cores, then just start 8 projects?

```
Copy
node index.js
```

This, ofcourse has a lot of problems

1. Just ugly to do this, keep track of the processes that are up and down
2. Processes will have port conflicts, you'll have to run each process on a separate port

This is where the `cluster module` comes into the picture

```
Copy
import express from "express";
import cluster from "cluster";
import os from "os";

const totalCPUs = os.cpus().length;

const port = 3000;

if (cluster.isPrimary) {
  console.log(`Number of CPUs is ${totalCPUs}`);
  console.log(`Primary ${process.pid} is running`);

  // Fork workers.
  for (let i = 0; i < totalCPUs; i++) {
    cluster.fork();
  }

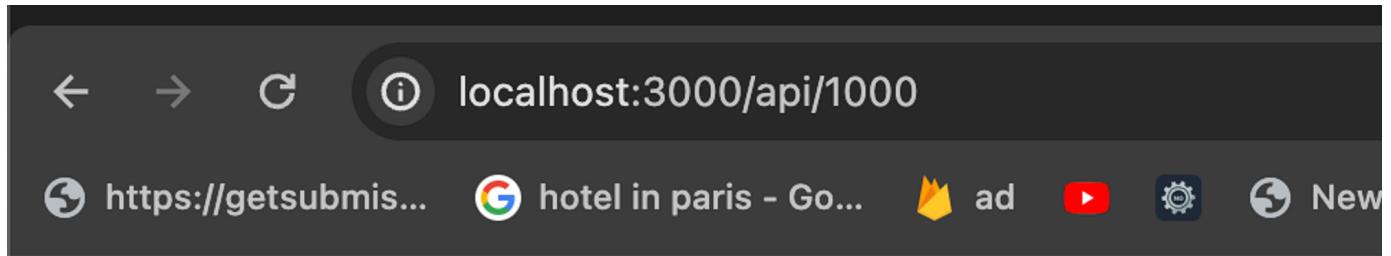
  cluster.on("exit", (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
    console.log("Let's fork another worker!");
    cluster.fork();
  });
}

cluster.on("listening", () => {
  console.log(`The ${process.pid} worker is listening on port ${port}`);
});
```

```
});  
} else {  
    const app = express();  
    console.log(`Worker ${process.pid} started`);  
  
    app.get("/", (req, res) => {  
        res.send("Hello World!");  
    });  
  
    app.get("/api/:n", function (req, res) {  
        let n = parseInt(req.params.n);  
        let count = 0;  
  
        if (n > 5000000000) n = 5000000000;  
  
        for (let i = 0; i <= n; i++) {  
            count += i;  
        }  
  
        res.send(`Final count is ${count} ${process.pid}`);  
    });  
  
    app.listen(port, () => {  
        console.log(`App listening on port ${port}`);  
    });  
}
```

Notice different pids in different devices

Browser



Final count is 500500 78331

Postman

A screenshot of the Postman application interface. The top bar shows the URL 'https://analytics.appx.co.in/others/get_total_course_and_series_sales?date=2024-04-12&database=tharunspeaks_db'. Below this, a 'GET' method is selected and the URL 'http://localhost:3000/api/1000' is entered. The 'Headers' tab is active, showing a list of selected headers:

Key	Value
Accept	application/json, text/javascript, */*; q=0.01
Accept-Language	en-GB,en-US;q=0.9,en;q=0.8
Connection	keep-alive
Cookie	ci_session=0k0cgknftskse2bv5o7a5k0hb8c

Below the headers, the 'Body' tab is selected, showing the response body: 'Final count is 500500 78330'. The status bar at the bottom right indicates 'Status: 200 OK' and 'Time: 4 ms'.

Capacity estimation

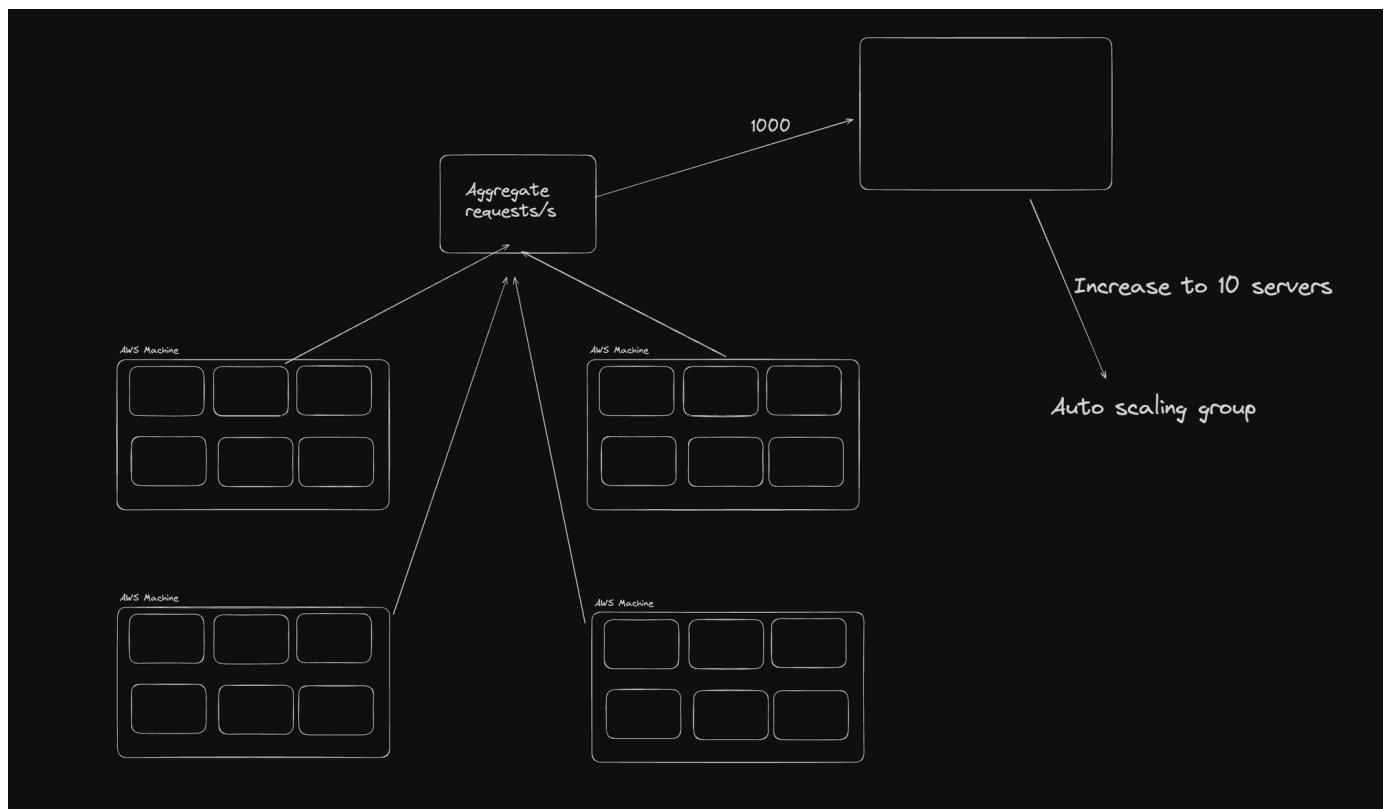
This is a common system design interview where they'll ask you

1. how would you scale your server
2. how do you handle spikes
3. How can you support a certain SLA given some traffic

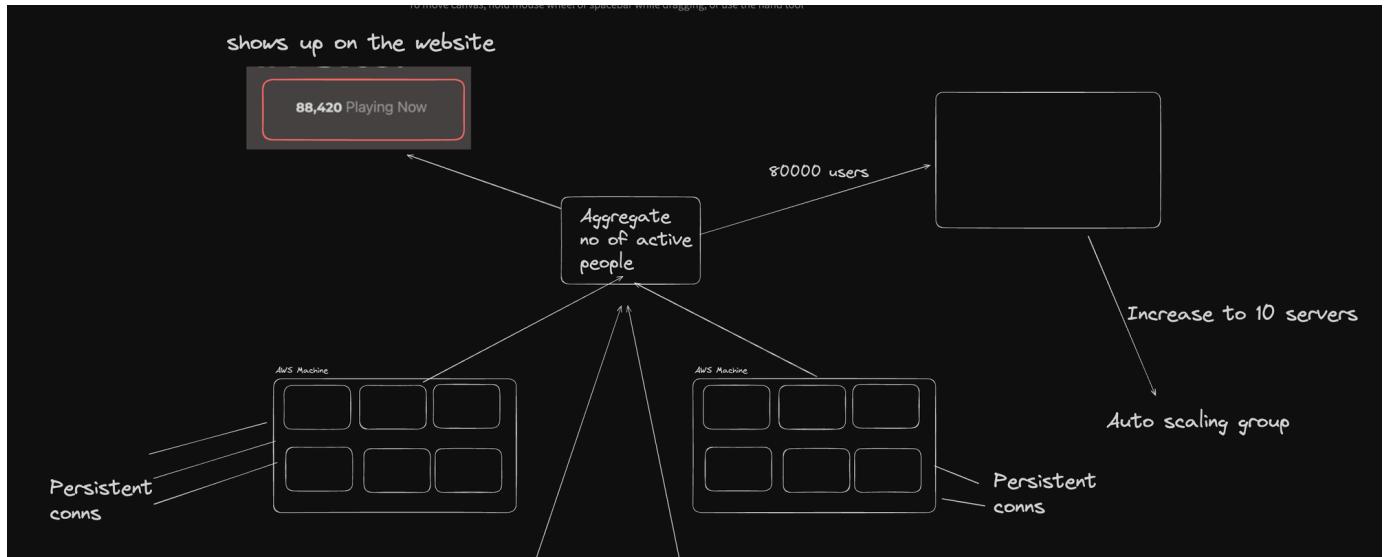
Answer usually requires a bunch of

1. paper math
2. Estimating requests/s
3. Assuming / monitoring how many requests a single machine can handle
4. Autoscaling machines based on the **load** that is estimated from time to time

Example #1 - PayTM app



Example #2 - Chess app



Horizontal scaling

Horizontal scaling represents increasing the number of instances you have based on a metric to be able to support more load.

AWS has the concept of **Auto scaling groups**, which as the name suggests lets you autoscale the number of machines based on certain metrics.

Buzz words

Images (AMI) - Snapshots of a machine from which you can create more machines

Load balancer - An entrypoint that your users send requests to that forwards it to one of many machines (or more specifically, to a target group). Its a **fully managed** service which means you don't have to worry about scaling it ever. AWS takes care of making it highly available.

Target groups - A group of EC2 instances that a load balancer can send requests to

Launch template - A template that can be used to start new machines



Please make sure you get rid of all your resources after this.

There are two ways you can use ASGs

- Create a EC2 instance.

1. install Node.js on it <https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-20-04>

2. Clone the repo - <https://github.com/100xdevs-cohort-2/week-22>

- Create an AMI with your machine
- Create security group
- Launch template
 - Ref for User data - <https://stackoverflow.com/questions/15904095/how-to-check-whether-my-user-data-passing-to-ec2-instance-is-working>

```
#!/bin/bash
export PATH=$PATH:/home/ubuntu/.nvm/versions/node/v22.0.0/bin/
echo "hi there before"
echo "hi there after"
npm install -g pm2
cd /home/ubuntu/week-22
pm2 start index.js
pm2 save
pm2 startup
```

Copy

The screenshot shows the AWS EC2 Launch Templates console. At the top, there's a navigation bar with 'Services' and a search bar. Below it, the URL is 'EC2 > Launch templates > Create launch template'. A green success message box says 'Success' and 'Successfully created node-app-1(lt-0e58c469517d669ab)'. Below the message, there's a 'Actions log' section. Under 'Next Steps', there are several options: 'Launch an instance' (with a note about On-Demand instances), 'Launch instance from this template', 'Create an Auto Scaling group from your template' (with a note about Auto Scaling), 'Create Auto Scaling group', 'Create Spot Fleet' (with a note about Spot Instances), and 'Create Spot Fleet'. At the bottom right of the main content area is a 'View launch templates' button.

- ASG
 - Callout on availability zones - ASGs try to balance instances in each zone

The screenshot shows the 'Network' configuration section of the AWS Auto Scaling group creation wizard. It includes fields for selecting a VPC and defining availability zones and subnets.

VPC: Choose the VPC that defines the virtual network for your Auto Scaling group. A dropdown menu shows 'vpc-089cbc9c696895a' with '172.31.0.0/16 Default' selected, with a refresh icon next to it. A 'Create a VPC' link is also present.

Availability Zones and subnets: Define which Availability Zones and subnets your Auto Scaling group can use in the chosen VPC. A dropdown menu shows 'Select Availability Zones and subnets' with a refresh icon. Two subnets are listed: 'ap-south-1a | subnet-0810906b79248efa6' (172.31.32.0/20, Default) and 'ap-south-1b | subnet-04411ee701286912f' (172.31.0.0/20, Default). Both have a delete icon (X) next to them. A 'Create a subnet' link is also present.

- Load balancer
 - Add an HTTPS Listener from your domain, request a certificate from ACM
- Target group - Attach the target group to the ASG

Autoscaling part

You can create an **dynamic scaling** policy

The screenshot shows the AWS Lambda function configuration page. At the top, there's a navigation bar with 'Lambda' and 'Options'. Below it, the path 'EC2 > Auto Scaling groups > node-app-1' is shown. The main title is 'Create dynamic scaling policy'. The form fields include:

- Policy type:** Target tracking scaling
- Scaling policy name:** Target Tracking Policy
- Metric type:** Average CPU utilization
- Target value:** 50
- Instance warmup:** 300 seconds
- Disable scale in to create only a scale-out policy:** (unchecked)

At the bottom right are 'Cancel' and 'Create' buttons.

Try playing with the Min and max on the ASG

The screenshot shows the AWS Auto Scaling group details page for 'node-app-1'. The top navigation bar includes 'Details', 'Activity', 'Automatic scaling', 'Instance management', 'Monitoring', and 'Instance refresh'. The 'Automatic scaling' tab is selected. The 'Group details' section contains the following information:

Auto Scaling group name	Desired capacity	Desired capacity type	Amazon Resource Name (ARN)
node-app-1	1	Units (number of instances)	arn:aws:autoscaling:ap-south-1:100749360009:autoScalingGroup:4ce2e6da-e011-44c8-8aa4-1f42afe1be0e:autoScalingGroupName/node-app-1
Date created Sun Apr 28 2024 15:01:23 GMT+0700 (Indochina Time)	Minimum capacity	Status	
	1	-	
	Maximum capacity		
	4		

Below this is a 'Launch template' section with an 'Edit' button.

Try killing servers

Try to stop a few servers in the ASG. Notice they spin back up to arrive at the desired amount.

Simulate a scale up

Try running an infinite for loop on the instance to see if a scale up happens

```
Copy
let c = 0,
    while (1) {
        c++;
    }
```

X ubuntu@ip-172-31-14-238: ~ (ssh) ...

Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.

See <https://ubuntu.com/esm> or run: sudo pro status

Last login: Sun Apr 28 08:04:47 2024 from 101.96.67.218

To run a command as administrator (user "root"), use "sudo <command>".

See "man sudo_root" for details.

```
ubuntu@ip-172-31-14-238:~$ ls
week-22
ubuntu@ip-172-31-14-238:~$ htop
ubuntu@ip-172-31-14-238:~$ vi index.js
ubuntu@ip-172-31-14-238:~$ node index.js
[]
```

X ubuntu@ip-172-31-14-238: ~ (ssh) ...

CPU[||||||||||||||||| 100.0%] Tasks: 44, 81 thr, 72 kthr; 1 running
Mem[||||||||| 362M/1.92G] Load average: 0.22 0.05 0.02
Swp[0K/0K] Uptime: 00:19:11

Main	I/O	PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
1690	ubuntu	20	0	975M	46848	41216	R	99.3	2.3	0:11.10	node index.js		
2013	ubuntu	20	0	8616	4224	3456	R	1.3	0.2	0:00.06	htop		
1	root	20	0	22244	13396	9556	S	0.0	0.7	0:05.68	/sbin/init		
138	root	20	0	66884	14636	13484	S	0.0	0.7	0:01.54	/usr/lib/systemd/s		
182	root	RT	0	282M	27136	8704	S	0.0	1.3	0:00.03	/sbin/multipathd -		
185	root	20	0	26456	8180	5108	S	0.0	0.4	0:00.25	/usr/lib/systemd/s		
187	root	20	0	282M	27136	8704	S	0.0	1.3	0:00.00	/sbin/multipathd -		
188	root	RT	0	282M	27136	8704	S	0.0	1.3	0:00.00	/sbin/multipathd -		
189	root	RT	0	282M	27136	8704	S	0.0	1.3	0:00.00	/sbin/multipathd -		
190	root	RT	0	282M	27136	8704	S	0.0	1.3	0:00.00	/sbin/multipathd -		
191	root	RT	0	282M	27136	8704	S	0.0	1.3	0:00.07	/sbin/multipathd -		

F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice -F8Nice +F9Kill F10Quit

You'll notice the desired capacity goes up by one in some time

Status	Description	Cause	Start time	End time
Not yet in service	Launching a new EC2 instance: i-011bf7ee8a3b91a34	At 2024-04-28T08:27:21Z a monitor alarm TargetTracking-node-app-1-AlarmHigh-57d88b00-496e-4af9-b4f4-d146173d2c8d in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 1 to 2. At 2024-04-28T08:27:29Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 2.	2024 April 28, 03:27:31 PM +07:00	

Try turning the infinite loop off and notice a scale down happens

Scaling via a Node.js app

Create a new user with permissions to `AutoscalingFullAccess`

The left screenshot shows the AWS Lambda console. A green notification bar at the top says "Public IPv4 DNS copied". Below it, a message says "ec2-13-233-67-81.ap-south-1.amazonaws.com (IP: 13.233.67.81)". A red arrow points from this message to the right screenshot. The right screenshot shows the AWS IAM User Management page for a user named "autoscaler". The "Permissions" tab is selected, showing an attached policy named "AutoScalingFullAccess".

```
import AWS from 'aws-sdk';

AWS.config.update({
  region: 'ap-south-1',
  accessKeyId: 'YOUR_ACCESS_KEY',
  secretAccessKey: 'YOUR_ACCESS_SECRET'
});

// Create an Auto Scaling client
const autoscaling = new AWS.AutoScaling();

// Function to update the desired capacity of an Auto Scaling group
const updateDesiredCapacity = (autoScalingGroupName: string, desiredCapacity: number) => {
  const params = {
    AutoScalingGroupName: autoScalingGroupName,
    DesiredCapacity: desiredCapacity
  };

  autoscaling.setDesiredCapacity(params, (err, data) => {
    if (err) {
      console.error(`Error updating desired capacity for ${autoScalingGroupName}: ${err}`);
    } else {
      console.log(`Desired capacity updated to ${desiredCapacity} for ${autoScalingGroupName}`);
    }
  });
}
```

```

        console.log("Error", err);
    } else {
        console.log("Success", data);
    }
});

};

// Example usage
const groupName = 'node-app-1'; // Set your Auto Scaling group name
const newDesiredCapacity = 3; // Set the new desired capacity

```

Indexing in Postgres

We've created postgres tables many times now. Let's see how/if indexing helps us speed up queries

- Create a postgres DB locally (don't use neon, we have a lot of data to store, will be very slow)

```
docker run -p 5432:5432 -e POSTGRES_PASSWORD=mysecretpassword -d postgres
```

Copy

- Connect to it and create some dummy data in it

```
docker exec -it container_id /bin/bash
psql -U postgres
```

Copy

- Create the schema for a simple medium like app

```

CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    name VARCHAR(255)
);
CREATE TABLE posts (
    post_id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL,
    title VARCHAR(255) NOT NULL,
    description TEXT,
    image VARCHAR(255),

```

Copy

```
FOREIGN KEY (user_id) REFERENCES users(user_id)
);
```

- Insert some dummy data in

```
DO $$  
DECLARE  
    returned_user_id INT;  
BEGIN  
    -- Insert 5 users  
    FOR i IN 1..5 LOOP  
        INSERT INTO users (email, password, name) VALUES  
        ('user'||i||'@example.com', 'pass'||i, 'User '||i)  
        RETURNING user_id INTO returned_user_id;  
  
        FOR j IN 1..500000 LOOP  
            INSERT INTO posts (user_id, title, description)  
            VALUES (returned_user_id, 'Title '||j, 'Description for post  
            END LOOP;  
        END LOOP;  
    END $$;
```

Copy

- Try running a query to get all the posts of a user and log the time it took

```
EXPLAIN ANALYSE SELECT * FROM posts WHERE user_id=1 LIMIT 1,
```

Copy

Focus on the **execution time**

- Add an index to user_id

```
CREATE INDEX idx_user_id ON posts (user_id),
```

Copy

Notice the **execution time** now.

What do you think happened that caused the query time to go down by so much?

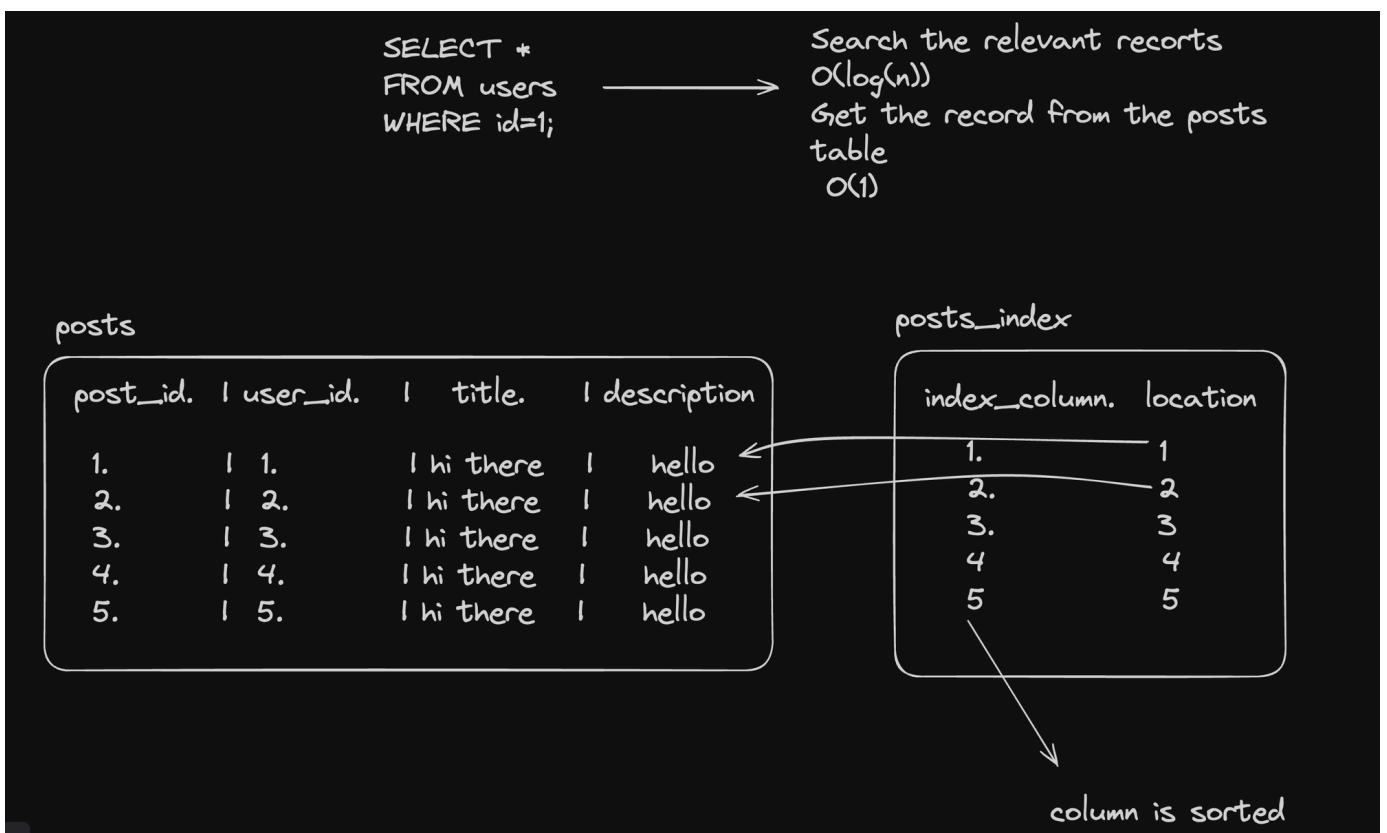
How indexing works (briefly)

When you create an index on a field, a new data structure (usually B-tree) is created that stores the mapping from the **index column** to the **location** of the record in the original table.

Search on the index is usually **$\log(n)$**

Without indexes

With indexes



The data pointer (in case of postgres) is the `page` and `offset` at which this record can be found.

Think of the index as the `appendix` of a book and the `location` as the `page + offset` of where this data can be found

Complex indexes

You can have index on more than one column for more complex queries

For example,

Give me all the posts of a user with given `id` with `title` "Class 1".

The index needs to have two keys now

```
CREATE INDEX idx_posts_user_id_title ON posts (description, title),Copy
```

- Try searching before the index is added and after it is added

```
SELECT * FROM posts WHERE title='title' AND description='my title',Copy
```

Indexes in Prisma

Ref - <https://www.prisma.io/docs/orm/prisma-schema/data-model/indexes>

You can add an index to a `model` in prisma by doing the following -

```
model User {
    id      String @id @default(uuid())
    username String @unique
    email   String @unique
    posts   Post[]
    createdAt DateTime @default(now())
    updatedAt DateTime @updatedAt
}

model Post {
    id      String @id @default(uuid())
    title  String
    content String?
    published Boolean @default(false)
    createdAt DateTime @default(now())
    updatedAt DateTime @updatedAt
    userId  String
    user    User    @relation(fields: [userId], references: [id])

    @@index([userId])
}
```

Copy

Let's look at daily code and see where all can we introduce an index

<https://github.com/code100x/daily-code/blob/main/packages/db/prisma/schema.prisma#L129>

Normalization

Normalization is the process of removing redundancy in your database.

Redundancy

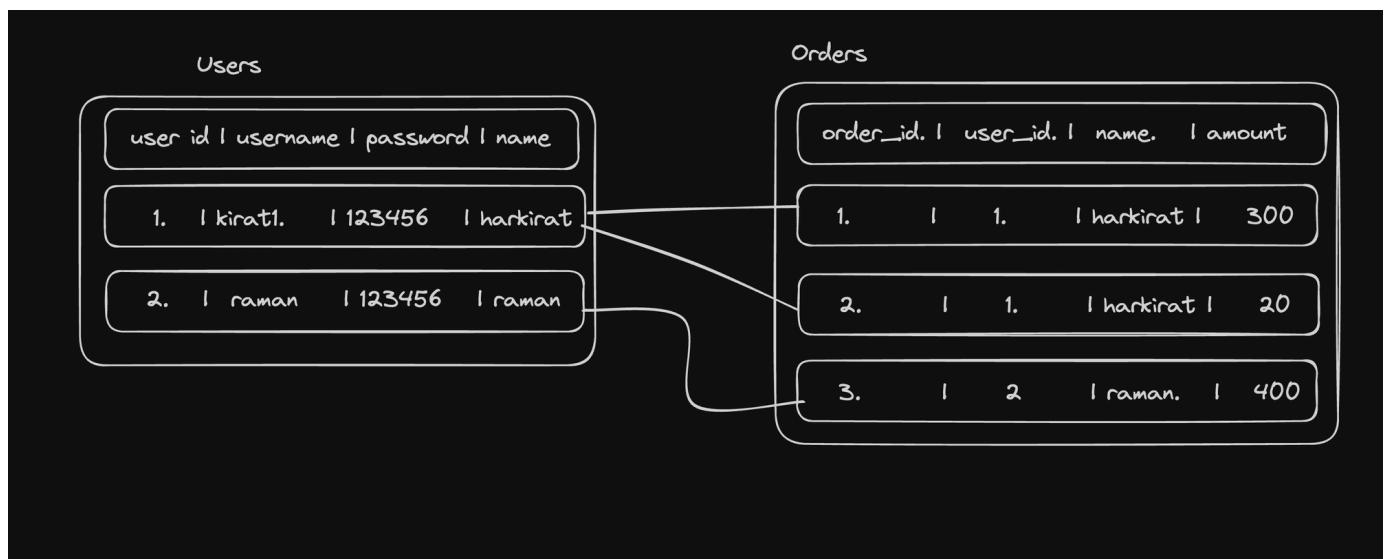
Redundant data means data that already exists elsewhere and we're duplicating it in two places

For example, if you have two tables

1. users

2. user_metadata

where you do the following -



If you notice, we've stored the name on the order in the Orders table, when it is already present in the Users table. This is what is **redundant** data.

Notice this schema is still **full proof**. We can get all the orders given a user id. We can tell the users details (username, name) given an order id.

Non full proof data

Users	Orders																								
<table border="1"> <thead> <tr> <th>user_id</th> <th>username</th> <th>password</th> <th>name</th> </tr> </thead> <tbody> <tr> <td>1.</td> <td>1 kirat1.</td> <td>1 123456</td> <td>1 harkirat</td> </tr> <tr> <td>2.</td> <td>1 raman</td> <td>1 123456</td> <td>1 raman</td> </tr> </tbody> </table>	user_id	username	password	name	1.	1 kirat1.	1 123456	1 harkirat	2.	1 raman	1 123456	1 raman	<table border="1"> <thead> <tr> <th>order_id</th> <th>name.</th> <th>amount</th> </tr> </thead> <tbody> <tr> <td>1.</td> <td>1 harkirat</td> <td>300</td> </tr> <tr> <td>2.</td> <td>1 harkirat</td> <td>20</td> </tr> <tr> <td>3.</td> <td>1 raman.</td> <td>400</td> </tr> </tbody> </table>	order_id	name.	amount	1.	1 harkirat	300	2.	1 harkirat	20	3.	1 raman.	400
user_id	username	password	name																						
1.	1 kirat1.	1 123456	1 harkirat																						
2.	1 raman	1 123456	1 raman																						
order_id	name.	amount																							
1.	1 harkirat	300																							
2.	1 harkirat	20																							
3.	1 raman.	400																							

This data doesn't have any relationship b/w Orders and users. This is just plain wrong. You can never tell the orders for a user (esp if 2 users can have the same name)

Types of relationships

Use case - Library management system

1. Users table
2. Library card table
3. Books table
4. Genre table

One to One

Each user has a single **Library card**

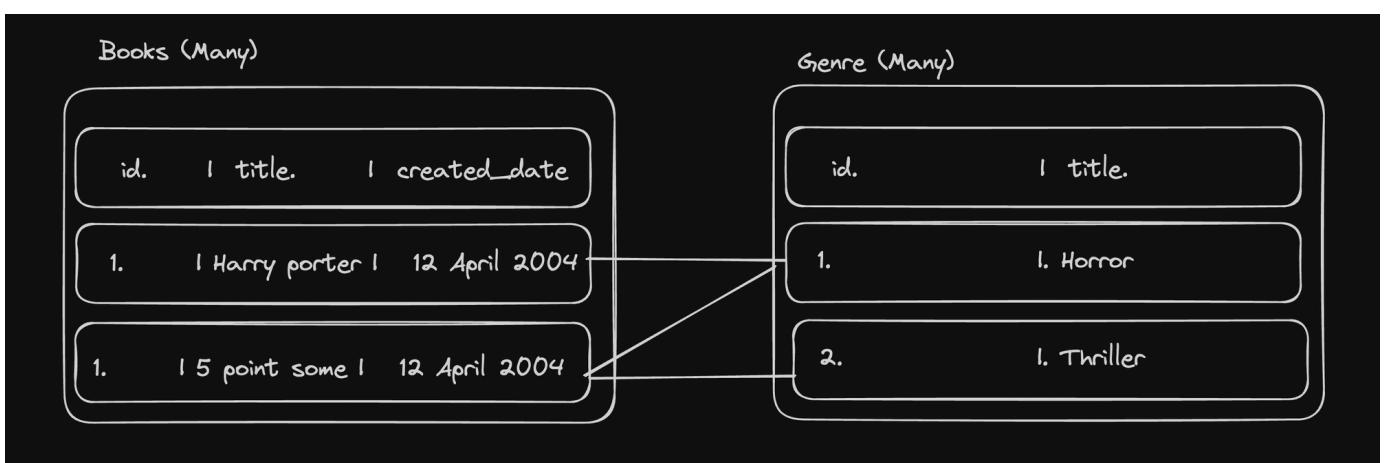


One to many

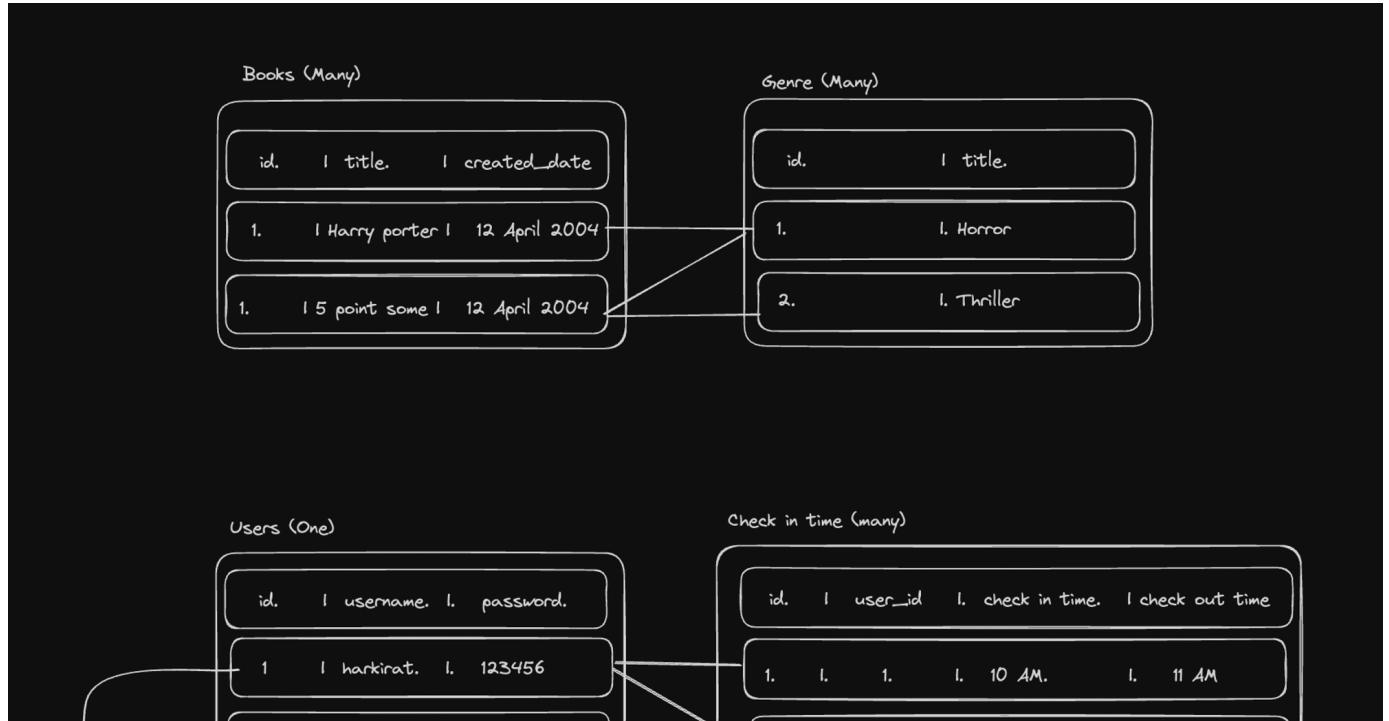
Many to one

Opposite of the thing above

Many to Many



Final graph



Normalizing data

Normalization in databases is a systematic approach of decomposing tables to eliminate data redundancy and improve data integrity.

The process typically progresses through several normal forms, each building on the last.

When you look at a schema, you can identify if it lies in one of the following categories of normalization

1. 1NF
2. 2NF
3. 3NF
4. BCNF
5. 4NF
6. 5NF

You aim to reach 3NF/BCNF usually. The lower you go, the more normalised your table is. But over normalization can lead to excessive joins

1NF

- **A single cell must not hold more than one value (atomicity):** This rule ensures that each column of a database table holds only atomic (indivisible) values, and multi-valued

attributes are split into separate columns. For example, if a column is meant to store phone numbers, and a person has multiple phone numbers, each number should be in a separate row, not as a list or set in a single cell.

StudentID	Name	Activities
1	John Doe	Basketball, Soccer, Chess Club
2	Emily White	Drama Club; Yearbook

- There must be a primary key for identification:** Each table should have a primary key, which is a column (or a set of columns) that uniquely identifies each row in a table
- No duplicated rows:** To ensure that the data in the table is organised properly and to uphold the integrity of the data, each row in the table should be unique. This rule works hand-in-hand with the presence of a primary key to prevent duplicate entries which can lead to data anomalies.
- Each column must have only one value for each row in the table:** This rule emphasizes that every column must hold only one value per row, and that value should be of the same kind for that column across all rows.

CustomerID	Name	ContactInfo
1	Alice Smith	alice@example.com, (555) 123-4567
2	Bob Johnson	bob@example.com; (555) 765-4321

CustomerID	Name	Email	Phone Number
1	Alice Smith	alice@example.com	(555) 123-4567
2	Bob Johnson	bob@example.com	(555) 765-4321

2NF

Ref -<https://www.studytonight.com/dbms/second-normal-form.php>

1NF gets rid of repeating rows. 2NF gets rid of redundancy

A table is said to be in 2NF if it meets the following criteria:

- is already in 1NF
- Has 0 partial dependency.

 Partial dependency - This occurs when a non-primary key attribute is dependent on part of a composite primary key, rather than on the whole primary key. In simpler terms, if your table has a primary key made up of multiple columns, a partial dependency exists if an attribute in the table is dependent only on a subset of those columns that form the primary key.

Example: Consider a table with the composite primary key (**StudentID**, **CourseID**) and other attributes like **InstructorName** and **CourseName**. If **CourseName** is dependent only on **CourseID** and not on the complete composite key (**StudentID**, **CourseID**), then **CourseName** has a partial dependency on the primary key. This violates 2NF.

Before normalization

Enrollments table

StudentID	CourseID	CourseName	InstructorName	Grade
001	CS101	Computer Science	Dr. Smith	A
002	CS101	Computer Science	Dr. Smith	B
001	MA202	Mathematics	Dr. Johnson	A
003	CS101	Computer Science	Dr. Smith	C
002	MA202	Mathematics	Dr. Johnson	B

Can you spot the redundancy over here? The instructor name and course name are repeated in rows, even though the name of an instructor should be the same for a given courseID

Primary key of this table is (student_id, course_id)

CourseName and InstructorName have a **partial dependency** on CourseID

Primary key

StudentID	CourseID	CourseName	InstructorName	Grade
001	CS101	Computer Science	Dr. Smith	A
002	CS101	Computer Science	Dr. Smith	B
001	MA202	Mathematics	Dr. Johnson	A
003	CS101	Computer Science	Dr. Smith	C
002	MA202	Mathematics	Dr. Johnson	B

After normalisation

CourseID	CourseName	InstructorName
CS101	Computer Science	Dr. Smith
MA202	Mathematics	Dr. Johnson

StudentID	CourseID	Grade
001	CS101	A
002	CS101	B
001	MA202	A
003	CS101	C
002	MA202	B

3NF

When a table is in 2NF, it eliminates repeating groups and redundancy, but it does not eliminate transitive partial dependency.

So, for a table to be in 3NF, it must:

- be in 2NF
- have no transitive partial dependency.

 A **transitive dependency** in a relational database occurs when one non-key attribute indirectly depends on the primary key through another non-key attribute.

For example

Department name has a transitive dependency on the primary key (employee id).

To normalise to 3NF, we need to do the following