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

```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?

```
node index.js
node index.js
node index.js
node index.js
node index.js
node index.js
node index.js
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 saparate port

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

```
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();
  });
```

```
} 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

Postman

Curl

Try to figure out why there is `stickiness` in the browser. Why the request from the same browser goes to the same pid

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
```

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

Try playing with the Min and max on the ASG

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

```
let c = 0;

while (1) {
    c++;
}
```

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

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`

```
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.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

// Call the function
updateDesiredCapacity(groupName, newDesiredCapacity);
```

Cleanup

Please delete all things one by one

1. ASG
2. Target group
3. Load balancer
4. Launch template
5. Image
6. Instance that the image was created from

Try using elastic beanstalk. Gives you the same benefits w/o the developer having to create all of these

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 (dont 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
```

- Connect to it and create some dummy data in it

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

- 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),
    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 '||j);
        END LOOP;
    END LOOP;
END $$;
```

- 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 5;
```

Focus on the `execution time`

- Add an index to user_id

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

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);
```

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

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

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])
}
```

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

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)

Normalisation is done on tables that are full proof to remove redundancy.

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.

- **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.

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

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 `CourserID`

After normalisation

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