Introduction

This report outlines the implementation of many critical features in the second phase of this project: an in-memory cache mechanism, a load balancing algorithm, servers replicas, and providing synchronization between these replicas through hooks.

Procedure:

1. We started off my modifying our frontend node by adding an in-memory cache to improve request processing latency. An in-memory cache object (cache) has been added to the application. This object stores data keyed by the search query. When a search is performed, the cache is checked first. If the data is present, it is directly used; otherwise, a new API call fetches the data and updates the cache.

```
const cache = {}; // In-memory cache object
```

```
try {
  if (cache[searchQuery]) {
    setSearchResults(cache[searchQuery]);
  } else {
```

```
cache[searchQuery] = response.data;
```

2. A load balancing algorithm has been implemented to evenly distribute requests among multiple replicas of the server. The Round Robin algorithm has been chosen for its simplicity and effectiveness.

here load balancing was added when we wanted to execute a search operation using the catalog server, the load will be distributed across the three catalog server replicas through round rubin's algorithm.

```
const servers = ["http://localhost:7000",
                 "http://localhost:6001",
                 "http://localhost:3000"];

export default function SearchAppBar({ setSearchResults }) {
  const [searchQuery, setSearchQuery] = useState('');

  useEffect(() => {
    const fetchData = async () => {
      try {
        if (cache[searchQuery]) {
          setSearchResults(cache[searchQuery]);
        } else {
          const response = await axios.get(servers[currentServerIndex] + '/api/catalog/search', {
            params: { query: searchQuery },
          });
          setSearchResults(response.data);
          cache[searchQuery] = response.data;
          currentServerIndex = (currentServerIndex + 1) % servers.length;
        }
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    };
```

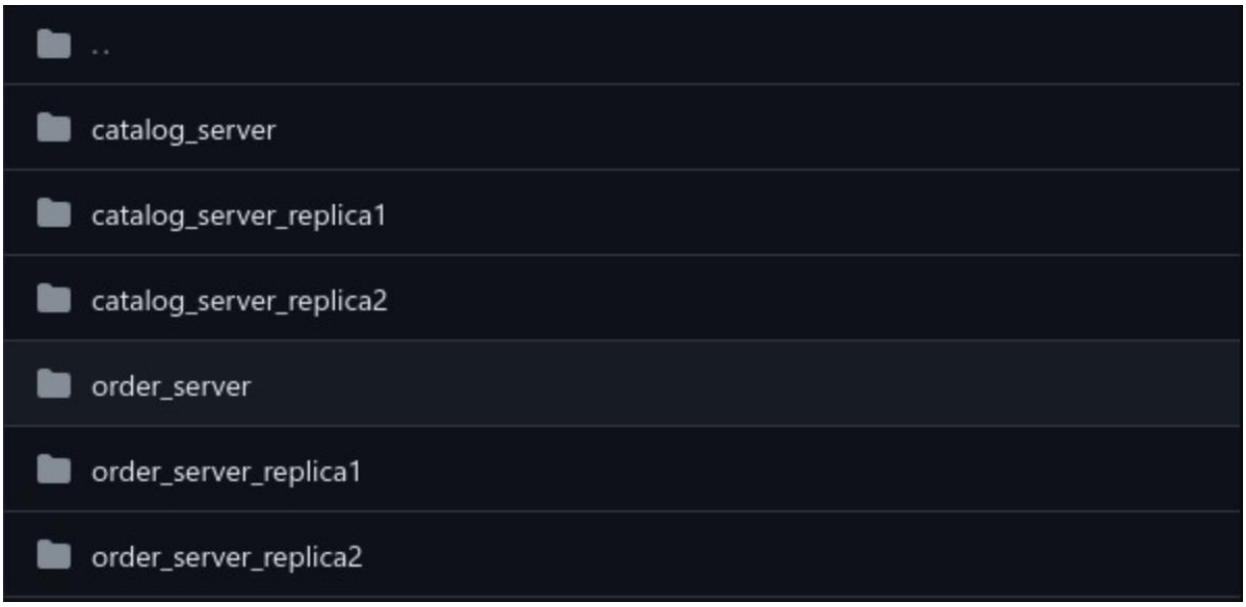
these are for the three replicas of the catalog server

The same concept is applied once again when retrieving books from the database and when purchasing a book through the order server.

```
const servers = ["http://localhost:3001",
                 "http://localhost:6001",          order server replicas
                 "http://localhost:7001"];
export default function BookCard(props) {
  const value = 4.5;
  const { id, title, price, cat, stock } = props;
  const [open, setOpen] = React.useState(false);
  const [purchaseStatus, setPurchaseStatus] = useState(null);

  const handlePurchase = async () => {
    try {
      const nextServer = servers.shift();
      servers.push(nextServer);

      const response = await axios.post(`${nextServer}/api/order/books/purch
        uuid: "25278d7b-7bc5-4b9e-8f8a-dc8df0944f4a",
        quantity: "1",
      });
      setPurchaseStatus({
        type: "success",
        message: nextServer+"Purchase successful!",
      });
    } catch (error) {
      console.error("Error making purchase:", error);
      setPurchaseStatus({
        type: "error",
        message: "Error making purchase. Please try again.",
```

```
let currentServerIndex = 0;
const servers = ["http://localhost:7000",
"http://localhost:6000",
"http://localhost:3000"];
function App() {
  // State to store the fetched data
  const [books, setBooks] = useState([]);
  const [searchResults, setSearchResults] = useState([]);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get(
          servers[currentServerIndex] + "/api/catalog/books"
        );
        setBooks(response.data);
        currentServerIndex = (currentServerIndex + 1) % servers.length;
        console.log(currentServerIndex);
      } catch (error) {
        console.error("Error fetching data:", error);
      }
    };

    fetchData();
```

3. Replicas of each server were created as you can see here



4. Basically what we did in this part was to ensure synchronization between these replicas we used hooks which is a class provided by the “Sequelize” library. These hooks are fired right before an insert, update or delete query happens. Through hooks requests will be sent to replicas that there’s a new update on the DB which ensure consistency and synchronization between all the servers/replicas. Here’s the flow of how this part works: When the leader server goes through DB updates/insertions/deletions it tells the rest of replicas to update their DBs accordingly. And to avoid concurrency/looping, when a replica itself goes through DB updates, it lets the leader server in on them and the

leader then communicates that to the rest of other replicas.
This function propagateChangesToReplicas was added to the DB to propagate the changes to the replicas according to the flow I've explained above.

```
const propagateChangesToReplicas = (changes) => {
  const replicaUrls = [
    "http://localhost:6000/api/sync/", // catalog replica 1
    "http://localhost:7000/api/sync/", //catalog replica 2
  ];

  // Propagate changes to each replica
  return Promise.all(
    replicaUrls.map(async (replicaUrl) => {
      await axios.post(
        replicaUrl,
        { changes },
        {
          headers: {
            "Content-Type": "application/json",
          },
        }
      );
    })
  );
};
```

Here we check for any previous insertions for the element in the DB to avoid looping, the same thing is done for the update case.

```
case "INSERT":
  // Check if the record already exists in the local database
  const existsInsert = await recordExists(data.uuid);
  if (!existsInsert) {
    await Books.create(data);
  } else {
    console.log(
      `Record with UUID ${data.uuid} already exists. Skipping...`
    );
  }
  break;
```

This endpoint was added to the route which basically does the synchronization between the replicas when changes to the local DB occur.

```
sync.post("/", async (req, res) => {
  const { changes } = req.body;

  try {
    // Apply changes to the local database
    await applyChangesToLocalDatabase(changes);

    res.status(200).json({ message: "Changes synchronized successfully" });
  } catch (error) {
    console.error("Error applying changes:", error);
    res.status(500).json({ error: "Internal server error" });
  }
});

export default sync;
```

To run the code all you need to is simply clone the repo into your device then hit this command **"npm install"** for the lead servers and replicas and the front end directory. You can run each server by "**node index.js"** meanwhile you can run the front-end part by executing **"npm run dev".**