

# IMDB MapReduce

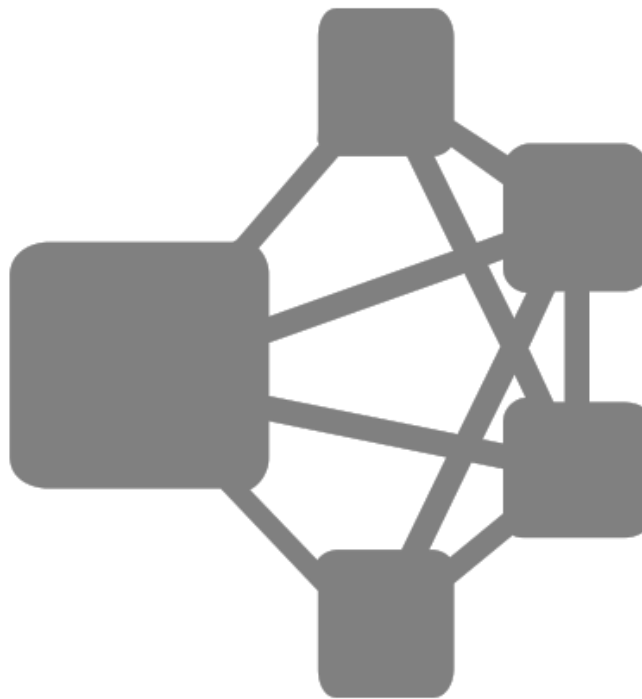
Final Project Report

Ben Gurion University of the Negev

Functional Programming in Concurrent and Distributed System

October 2024

Instructors – Dr Yehuda Ben Shimol.



**Haim Fellner Cohen**

### **Abstract:**

In this project I implemented a map reduce system. The system includes three entities:

- Master
- Client
- Server

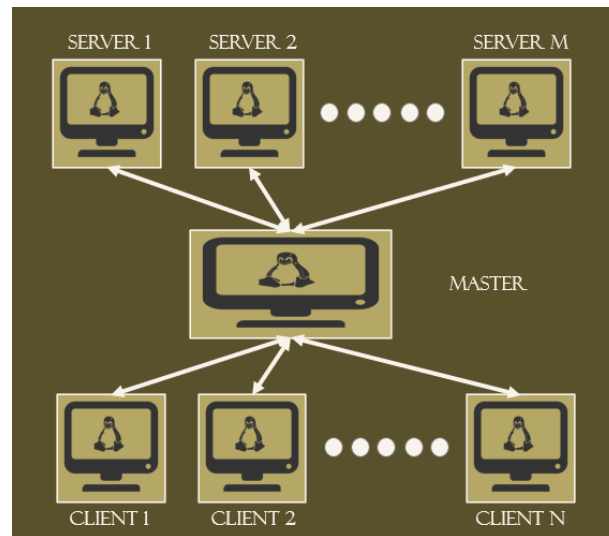
Using Master entity who control on the queries, data and merging all together. All servers all located in different computers as well as the master and the client.

All the clients available on the graphic user interfaces.

### **Map Reduce:**

MapReduce is like a team effort for handling big chunks of data. When we are working a lot of information that needs sorting and summarizing. The Map phase is where the work begins: each team member takes a piece of the data, processes it, and turns it into small, manageable key-value pairs. Then, in the Reduce phase, all those pairs are gathered together, and the team collaborates to combine them, making sense of the information and producing a final result. This method not only speeds up the process by working in parallel but also ensures that if something goes wrong, the system can recover and keep going. It's a smart way to tackle large-scale data challenges!

## System:

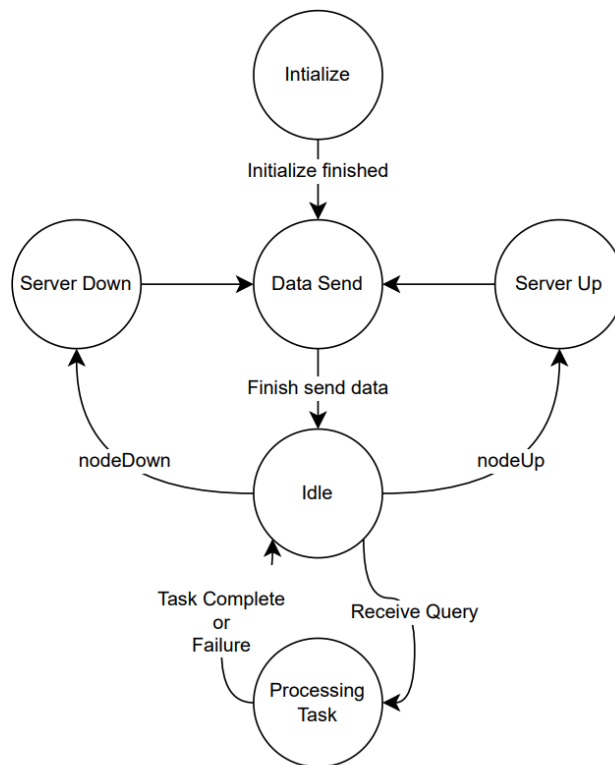


Master: The master manages all interactions from clients and servers, handling each request with a dedicated process.

Multi-Server: Each server receives an equal amount of the data and executes the Map-Reduce algorithm as needed, based on the client's query.

Multi-Client: Each client can submit queries to the master and receive targeted responses, displayed in the client's GUI as graph.

## Master:



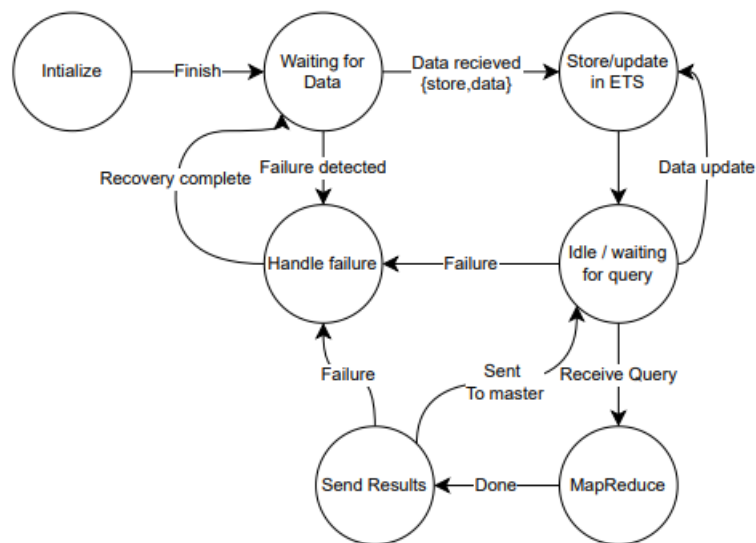
The Master module starts in the Initialization State, where it loads data from csv file into an ETS table and sets up node monitoring. Upon completing initialization, it transitions to the Data send State. In this state, the master sends data to all available servers by split and shuffle, this ensures that each server has the necessary data to process queries.

Once data distribution is complete, the master moves to the Idle State, ready to handle client queries and monitor node events. When a client query is received (`handle_call(Query, From, State)`), the master transitions to the Processing Task State. Here, it spawns a new process to handle the task, distributes the query to available servers, gathers the results, merges and processes them, and finally sends the result back to the client. After the task is completed, it returns to the Idle State.

If a server node goes down (detected via `handle_info({nodedown, Node}, State)`), the master transitions to the Handling Node Down State. It updates the list of available servers and moves back to the Data send State to redistribute data among the remaining servers,

ensuring data consistency and availability. Similarly, when a new server node comes up (`handle_cast({nodeup, Node}, State)`), the master transitions to the Handling Node Up State, updates the server list, and re-enters the Data Distribution State to send data to all servers, including the new one. After data is sent, the master returns to the Idle State, ready for new queries or node events.

### Server:



The Server module begins in the Initialization State, where it initializes its internal state, reads client nodes from txt file", and notifies the master of its availability using `gen_server: cast({masterpid, list_to_atom(MasterNode)}, {nodeup, node()})`. After initialization, it transitions to the Waiting for Data State, indicating it is ready to receive data from the master.

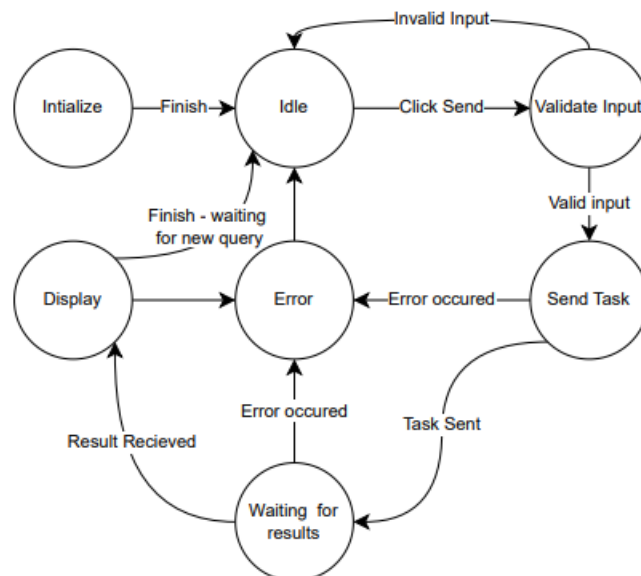
Upon receiving data (`handle_cast({store, Data}, State)`), the server transitions to the Data Stored State. In this state, it stores the data into an ETS table and updates its state variables accordingly. When a query is received from the master or a client (`handle_call(Query, From, State)`), it moves to the MapReduce State. The server processes the query using MapReduce by spawning a new process which performs the computation.

Once the MapReduce operation is complete, the server transitions to the Sending Results State. Here, it sends the computed results back to the requester (`FromPID ! Result`).

After successfully sending the results, the server enters the Idle State, where it waits for new queries or data updates. If new data arrives during this time, it transitions back to the Data Stored State to update its data store.

In case of a failure, such as data corruption or the need to re-receive data, the server transitions from any current state to the Handling Failure State. During failure handling, it performs necessary recovery actions, which may include re-initialization or requesting data from the master again. Once recovery is complete, it moves back to the Waiting for Data State to receive new data, ensuring it can resume normal operations.

### Client:



The Client module starts in the Initialization State, where it sets up the graphical user interface components. After the GUI is fully initialized, it transitions to the Idle State, awaiting user interaction.

When the user inputs a search value, selects a search method, and clicks the send button, the client transitions to the Validating Input State. Here, it checks if all necessary inputs are provided and valid. If the input is invalid, it displays an error message and returns to the Idle State for the user to correct their input. If the input is valid, it moves to the Sending Query State.

In the Sending Query State, the client constructs a record with the user's input and sends it to the master server. It also spawns a process to monitor the master server's status. After sending the task, the client transitions to the Waiting for Result State, where it waits for the master server's response.

Upon receiving the result from the master, the client transitions to the Display State. It processes the result by generating a graph using the appropriate module (movie or actor), and displays the graph to the user within a new window. After displaying the result, it returns to the Idle State, ready for the next user query.

If an error occurs at any point during the sending or waiting process—such as the master server being offline or a communication failure—the client transitions to the Error State. In this state, it displays an error message to the user and performs any necessary cleanup, such as terminating monitoring processes. Once the error is acknowledged, the client returns to the Idle State, allowing the user to attempt another query or exit the application.

### **Includes:**

In the project I used some tools and libraries in erlang that help me to work on the project.

- Erlang OTP 24.
- Graphics – I used GraphViz in order to implement visual graphs. In order to do so, I exported the final data in DOT file and crated image using GraphViz.
- Erlang Term Storage (ETS)- is a powerful Erlang feature for creating and handling in-memory tables to store Erlang terms. It allows data to be stored and accessed quickly, making it ideal for efficient data manipulation in concurrent applications. I used ETS for many reasons – First, in order to store all the data in the master to be able to send actor list by movie name (very quick because based on hash table. Second, all server keeps the data he gets in ETS in order to search values quickly.
- The OTP framework in Erlang includes a built-in module called `gen_server`, which provides a standardized way to handle messages and initialize objects. In simple terms, it helps manage the logic for processes that need to communicate and keep track of their state. In my project, master and server are set up to use this `gen_server` behavior, meaning they follow a specific structure for handling messages and actions consistently.



## **Statistics:**

In this phase I focus on analyzing performance and efficiency across different search types and system configurations. In the beginning I compared between 3 PCs and 5 PCs in order to check if more servers (which means more process power) results better results. In this experiment I have seen that as long as I use more servers, I will get faster results.

In addition, I measured performance consistency and variation. Based on 20 different search inputs for both actor and title methods, data was collected on running time per query, average running time, and standard deviation for each PC. System-wide metrics, including the overall average and standard deviation across all PCs, were calculated to evaluate consistency. These metrics highlight the reliability and scalability of the system when handling multiple requests in a distributed environment.

Each PC uses all the cores he has (14 in my case) and 60 process – more servers provide more processes.

## **Instructions:**

- Make sure servers and clients txt files are updated with the correct data.

Example – <Name of entity>@<IP>

Make sure that the master is on the first line on clients.txt.

### 1. Compile all the files:

```
erl  
  
c(actor_graph). c(clientGUI). c(csv_to_ets). c(dataToServers). c(mapReduce).  
c(mapReduce). c(master). c(movie_graph). c(parse_csv). c(server).
```

### 2. Run servers:

```
erl -name <serverName@IP> -setcookie <cookieName> -run server start_link
```

- a. Make sure the serverName and its IP pairs are in the servers.txt file
- b. Choose your own cookieName – this cookie serve all the entities.

### 3. Run master:

```
erl -name <masterName@IP> -setcookie <cookieName> -run master start_link
```

- a. Make sure the masterName and its IP is the first line in clients.txt file and there is only one.
- b. Make sure all the servers finish receives and process data.
- c. Make sure master finish to create ETS file.


### 4. Run clients:

```
erl -name <masterName@IP> -setcookie <cookieName>  
  
clientGUI:start().
```

- a. Make sure the clientName and its IP pairs are in the clients.txt file start from second row.

## GUI instructions:

Final Project - MapReduce



**Select Method:**

Title

Actor

בחירת שיטת החיפוש

**Insert Movie or Actor:**

חיפוש הנדרש

Send value to search

הרצת חיפוש